1999

Robot navigation using ultrasonic feedback

Akihiko Baba
West Virginia University

Robot Navigation using Ultrasonic Feedback

Akihiko Baba

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirement
for the degree of

Master of Science
in
Mechanical Engineering

Larry E. Banta, Ph.D., Chair
John E. Sneckenberger, Ph.D.
Powsiri Klinkhachorn, Ph.D.

Department of Mechanical and Aerospace Engineering

Morgantown, West Virginia
1999

Keywords: Mobile Robot, Navigation, Ultrasonic Sensors

ABSTRACT

Robot Navigation Using Ultrasonic Feedback

Akihiko Baba

This thesis presents an autonomous navigation program for West Virginia University's mobile robot platform ANT. The program allows the ANT to find a predefined goal by comparing sensor data with a given map of the environment. ANT's ultrasonic sensors were used to sample the surroundings continuously, and this information was combined with dead reckoning data to build a two-dimensional map of the environment. Borenstein's Histogramic In-Motion Mapping technique was used to build sensor maps, which were analyzed to find features such as walls and corners. The features were matched with features in the given map to correct dead reckoning errors. Partial success was demonstrated in this work. However it was concluded that more than seven sensors are necessary when operating in an environment with smooth walls. The failure modes for the robot were analyzed and suggestions are made to overcome some of the sensing problems.

Acknowledgment

I would like to express my appreciation to many individuals who have been very helpful in the completion of this thesis. I would like to thank Dr. Larry E. Banta acting as my research advisor. His patient, continuous help and his permission for me to use the mobile robot testbed ANT for this research, were greatly appreciated and necessary to the completion of this project. I am also grateful to my other committee members, Dr. John E. Sneckenberger and Dr. Powsiri Klinkhachorn for the guidance.

Additionally, I would like to deeply thank to my family for their patience and support in this very difficult journey of the last eight years since I came to this country. I also would like to express special thanks to my girlfriend Akiko Okada and my good friends Hiroko and Judson Mabee.

Table of Contents

	Page No.
Title Page	i
Abstract	ii
Acknowledgment	iii
Table of Contents	iv
List of Figures	vii
Chapter 1: Introduction	1
1.1 Problem Statement and Motivation	1
1.2 Research Objectives	3
Chapter 2: Literature Review	4
2.1 Motion of Tracked Vehicle	4
2.2 Ultrasonic Sensors	6
2.3 Mapping Technique	8
2.4 Navigation	9
Chapter 3: The Testbed Mobile Robot ANT	13
3.1 Robot Geometry	14
3.2 Motor Control	15
3.3 Ultrasonic Sensor System	16
3.4 Robot Hardware Control System	18
Chapter 4: Technical Approach	19

4.1 Optical Encoders 19

4.2 Ultrasonic Sensors 21

4.3 Mapping Technique 24

4.4 Feature Recognition and Map Matching 29

4.4.1 Finding Walls 31

4.4.2 Finding Corners 33

4.5 Navigation 35

Chapter 5: Experimental Results 37

5.1 Experimental Method and Setup 37

5.2 Map Building 39

5.3 Feature Recognition and Map Matching 42

5.4 Navigation 44

5.5 Failure Case Analysis 46

5.6 Recalibration of Ultrasonic Sensors 50

5.7 Navigation Retest 52

Chapter 6: Conclusion and Future Study 53

6.1 Conclusion from the Experimental Results 53

6.2 Suggestions for Future Study 55

References 57

Appendix

A Equipment List 60

A.1 List of the Equipment Used in the Robot 61

B Results of Ten Test Runs 63

B.1 Results from Test Run #1 64

B.2 Results from Test Run #2 65

B.3 Results from Test Run #3 66

B.4 Results from Test Run #4 67

B.5 Results from Test Run #5 68

B.6 Results from Test Run #6 69

B.7 Results from Test Run #7 70

B.8 Results from Test Run #8 71

B.9 Results from Test Run #9 72

B.10 Results from Test Run #10 73

C Source Codes 74

C.1 Navigation Program 75

C.2 Header File included in the Navigation Program 105

C.3 Manual Control Program 108

C.4 Header File included in Manual Control Program 120

List of Figures

		Page No.
Figure 3-1	The Mobile Robot Testbed ANT	13
Figure 3-2	Schematic Drawing of the Mobile Robot Testbed ANT.	14
Figure 3-3	Schematic Diagram of the ANT's Control System	17
Figure 4-1	Drawing of Motor Shaft and Optical Sensors	19
Figure 4-2	Graph of the 90 degrees Phase Difference	20
Figure 4-3	Calibration Curve for Ultrasonic Sensors	22
Figure 4-4	Firing Order of Ultrasonic Sensors	23
Figure 4-5	Two-dimensional Projection of the Conical Field of an Ultrasonic Sensor	24
Figure 4-6	Updating the Histogram Map	26
Figure 4-7	Given Map in a Matrix Form	27
Figure 4-8	The Global and Local Coordinate Frames	28
Figure 4-9	An Example of Mapping Procedure	29
Figure 4-10	Local Sensor Map Filtering Procedure	30
Figure 4-11	Flow Chart of the Obstacle Grouping Algorithm	31
Figure 4-12	Flow Chart of the Wall Matching Algorithm	32
Figure 4-13	Features of a Corner	33
Figure 4-14	Path Planing	35

Figure 5-1	Actual Path of ANT of Test Run #1	37
Figure 5-2	Sensor Map from Test Run #1	39
Figure 5-3	Sensor Map and Actual Path from Test Run #2	39
Figure 5-4	Sensor Maps and Actual Paths from Test Run #5 and #8	40
Figure 5-5	Error Plots from Test Run #5 and #8	41
Figure 5-6	An Example of Map Matching Procedure from Test Run #1	43
Figure 5-7	Actual Path of Test Run #1	44
Figure 5-8	Error Plot from Test Run #1	44
Figure 5-9	Actual Paths of Test Run #3 and #10	46
Figure 5-10	Local Map at the Second Turning Point from Test Run #5	47
Figure 5-11	Actual Path of Test Run #5	47
Figure 5-12	Turning Radius of ANT	48
Figure 5-13	Actual Path of Test Run #7	48
Figure 5-14	Ultrasonic Sensor Reading Distribution	50
Figure 5-15	New Calibration Curve of the Ultrasonic Sensors	51

Chapter 1

Introduction

1.1 Problem Statement and Motivation

Today, autonomous mobile robots are available for well-structured environments such as factory warehouses or office buildings. Automobile companies are developing auto-drive cars which drivers can recline the seat and relax while the car is running on a highway. Accurate positioning and collision-free navigation are always the most important issues in mobile robotics [1]. Although they are called autonomous, most of the automated carts which are currently available require their environment to be modified; for example, the floor must be painted to guide them from one location to another [2]. Modification to the environment is not very effective both practically and economically, i.e. the floor must be repainted every time to change the robot path. In order to eliminate the environmental modification, it is necessary for the robot to build maps using various sensors to detect walls, landmarks, and obstacles surrounding the robot, and dead reckoning which is defined as a navigation depends on distances traveled, to navigate itself to a goal successfully.

A dual-tracked vehicle is adequate to traverse not only rough surfaces but also narrow paths because of its ability to turn on its axis. Each track has one degree-of-freedom which means that it can only move along one axis, back and forth. To make a turn, each track must slip a certain amount so that the difference between two tracks turns the vehicle.

This thesis presents a vehicle navigation approach using the concept of Histogram Mapping Technique developed by Borenstein [7], [9], [11], [12]. Seven Panasonic Ultrasonic Proximity Transducers are utilized to obtain environmental information, and optical encoders on each track of the vehicle to compute the distance traveled. Using the ultrasonic data and dead reckoning, a robot is able to construct an environmental map which can be compared with a given map to locate itself.

Ultrasonic sensors are widely used as a distance measuring device because they are inexpensive and easy to use [8]. It is obvious that the more sensors the robot has, the more precise the map should become. However, ultrasonic sensors can return false readings easily when the number of the sensors and sampling rate increase, which is known as *cross talk*.

This thesis presents an off-line navigation program for a dual-tracked mobile robot. A map of the environment, the destination, and turning points are given to the program in advance of the operation. Real-time obstacle avoidance and path planning were not part of this work.

1.2 Research Objectives

The objective of this thesis is provide the West Virginia University's mobile robot platform ANT III (Autonomous Navigation Testbed III) with an autonomous navigation program to achieve a predefined goal by comparing the sensor data with a given map of the environment. This would be accomplished by using ANT's ultrasonic sensors to sample the surroundings continuously, and combining this information with the dead reckoning data to build a two-dimensional environmental map.

The specific research objectives of this thesis are;

1. To integrate the current ultrasonic sensor data with the dead reckoning data to build an updated two-dimensional environmental map which is called the sensor map.

2. To analyze and compare the sensor map with the given modeled terrain map. Two local maps, which have a smaller size than that of the entire maps, surrounding the robot are extracted from both the model and sensor maps for the comparison.

3. To navigate the robot through a hallway assuming that the walls appear only horizontal and vertical planes, and no unexpected obstacles exist.

Chapter 2

Literature Review

A review of literature has been performed for this research in the field of mobile robot path planning, mobile robot navigation, mobile platforms, sensor fusion, motion of tracked vehicles, noise reduction methods for ultrasonic sensors, and mapping methods for mobile robots. This chapter provides an overview of research previously conducted, dividing it into four categories; motion of tracked vehicles, ultrasonic sensors, mapping methods, and navigation techniques.

2.1 Motion of Tracked Vehicle

The track was first developed in the 18th century and widely used for off-road transportation. Due to the fact that the tracks have only one degree of freedom and have to slip to make a turn or curved path, the analysis of the tracked vehicle is so complicated that designs had been done by "trial and error" and there had been no general equations to express the behavior of the tracked vehicle. In other words, tracks depend heavily on the conditions of the surface such as friction or roughness [5].

Kitano [3], [4] has done significant studies on the tracked vehicle. He stated that "Stationary movements are rather unusual and almost all motions on a tracked vehicle are considered non-stationary." He developed highly mathematical models of the tracked vehicle and of tracked vehicle maneuverability on level ground. The non-stationary curvilinear motion of tracked vehicles makes the calculation complex and they use a

numerical analysis of their differential equations on a computer. The computation takes velocities of both left and right drive sprockets as input, and calculates track slip velocities, side slip angles, yaw rate, and acceleration of the center of gravity. Since both of the tracks have only one degree-of-freedom, that is back and forward motion, in order to make turns they introduce slippage of the tracks. They also include weight distribution along the track, friction forces on tracks, and rolling resistance into their consideration. His model is applicable for relatively fast moving vehicles such as military tanks, but not for slow moving vehicles such as a coal miner in which inertial effects are negligible.

Acker and Krishnan [6] presented a mathematical model of the turning motion of the two tracked miner. The turning radius is derived as a function of tread speeds, the center of gravity location, and the friction characteristic of the driving surface. It is assumed that the vehicle accelerates to its maximum maneuvering speed within a few inches, and inertia is thus ignored. Thus the trajectory is independent of the mass of the vehicle and only depends on relative speeds of both tracks. The instantaneous velocity of each segment of the track is derived as a resultant of two components, the velocity of the vehicle frame at that location relative to the ground, and the velocity of the segment relative to the vehicle frame. While turning, all segments in contact with the operating surface slide. The slippage is computed from experimental data and a plot of turning radius vs. ratio of the tread speed is produced.

Banta [21] introduced a dead reckoning technique for a vehicle using a trapezoid integration rule to estimate the robot's location. The trajectory can be approximated as a series of straight line segments with length S, with an angular change of θ between them.

He defined a new variable ϕ which is half of the angle for notational convenience. As the vehicle makes a smooth curve, the position can be estimated as a straight line with an angle of half of the ending angle. It is formulated as:

$$x_i = x_{i-1} - S_{i-1}\sin(\theta_{i-1} + \varphi_{i-1})$$
$$y_i = y_{i-1} - S_{i-1}\cos(\theta_{i-1} + \varphi_{i-1})$$
$$\theta_i = \theta_{i-1} + 2\varphi_{i-1}$$

where the subscript i is a discrete time sample index. This estimation is effective in a case for which the robot is moving slow enough compared to the encoder sampling period.

Tenney [20] studied course correction for a tracked vehicle. Because of constraint of the tracked vehicle, that the motion of the tracked vehicles depends on both longitudinal and lateral friction forces on the track, it is assumed that the friction characteristic is constant throughout the operating surface. A plot of turning radius vs. ratio of inner and outer tread speed for WVU's mobile robot is made experimentally. With that calibration and the initial conditions including the initial position and orientation the desired trajectory, the control algorithm is developed to return smoothly to the desired trajectory using optimal control techniques.

2.2 Ultrasonic Sensors

Ultrasonic range sensors are widely used in mobile robot applications because of their low cost and ease of use [8]. By placing them around the robot at some interval, they can give an omnidirectional coverage with enough overlaps of each sensor to overcome the poor directionality. Joong Hyup Ko [18] et al. studied echo amplitude patterns of ultrasonic sensors statistically and analyzed the relationship between the repulsive echo

and the orientation of the sensor.

Since ultrasonic sensors fire an ultrasonic pulse and measure the time until the bounced pulse returns, multiple sensors working in same environment can cause noise. Borenstein [7] categorized two cases which cause ultrasonic sensors to give false readings. One is environmental noise from other ultrasonic sensors which is very likely to occur when multiple robots equipped ultrasonic sensors are in the same environment. The other is internal noise from onboard ultrasonic sensors, which is known as crosstalk.

According to Borenstein [7], [9], one method to eliminate simple noise is to compare the readings. It can be said that the "good" readings should be in some range while the robot is traveling slow enough compared to the sampling period. This is an effective way but it doesn't work for the "crosstalk" errors because they are somewhat identical. The sensors are actually reading the echoes even though they are fired from other sensors. So the other way is to give some combination of delays between each firing so that echoes from others cannot be received. The time table becomes more complex as the number of ultrasonic sensors increases. This method helps improve the sensor performance.

Joong Hyup Ko, et al. [18] proposed a method to extract multiple acoustic landmarks at the same time for mobile robot indoor navigation. They discussed how a single ultrasonic sensor works and expand it to multiple sensors to find vertical walls to construct a local map, then compare it to the known global map. They used the echo amplitude pattern (EAP) to measure the orientation of the obstacle with respect to a single sensor. They combine data obtained by multiple sensors to calculate the distance and

orientation to an obstacle.

2.3 Mapping Technique

Mapping techniques using ultrasonic range sensors have been studied and modified because it is the most essential capability for a mobile robot to constantly update the environmental map internally to locate itself and successfully navigate autonomously [14]. The most important issue is how to deal with the uncertainty and noise from the ultrasonic sensors.

One of the popular methods is edge-detection [12], because vertical edges are easily picked up by the ultrasonic sensors. It is also true for vision systems [10]. This algorithm tries to locate vertical edges and connect them to find an obstacle. Disadvantages of this method are that a robot must stop in front of an obstacle to gather the sensor information in order to perform this task, and the poor directionality of the sensor gives uncertainty in the edge location. The ultrasonic sensors have a conical field of view, typically enclosing a 30 degree angle. Because of this nature, there is no way to determine the bearing angle from the sensor to the object. Additionally, this method does not deal with misreadings [12].

A pioneering method which represents obstacles in certainty-grid model was developed at Carnegie Mellon University [11]. In this method, the information from different kinds of sensors are combined into one certainty map. This early method has a disadvantage that it requires the vehicle to stop to collect the sensor data.

Other methods have been developed to include probability. In these algorithms,

map grids contain a certainty value for each grid. Byung-Kwon Min, et al. [8] developed a probability map in which each cell has a probability between zero and unity. In the beginning, each cell starts with uncertainty value of 0.5. The probability map is updated constantly according to the sensor readings using the Bayesian conditional probability formula. This method requires complex computation each time.

Borenstein, et al. introduced a more advanced technique called Histogramic In-Motion Mapping (HIMM) for a mobile robot in motion. This method requires the vehicle to keep moving to build a map. Both the CMU and HIMM systems use two-dimensional Cartesian histogram grids to represent a map. The major difference between them is that HIMM only counts one cell on the acoustic axis of the ultrasonic sensor when the sensor detects an obstacle, while the CMU concept counts all the cells in the range which is usually a 30 degree angle for a Polaroid ultrasonic sensor. Although the HIMM method is less accurate if the robot is stationary, it works better and needs less computational effort when the robot is traveling.

Gonzalez, et al. [13] used statistical analysis on the ultrasonic sensor to evaluate how the distance and angle to an obstacle affects the readings and noise. They then modified Borenstein's Vector Field Histogram by giving some probabilities on the arc. The closer to the acoustic axis, the bigger the certainty becomes.

2.4 Navigation

Mobile robot navigation is divided into two categories, off- and on-line. Off-line navigation means that the map is given and path planning is done a priori, and local

navigation is used to avoid unexpected obstacles . The latter, on-line navigation, can perform guidance in partially known or completely unknown environments. The concept of on-line navigation must include real-time path planning.

L. D. Seneviratne, et al. [1] introduced a simple path planning method based on triangulation. A triangulation graph connecting the edges of the polygons should be constructed, first. The free space is decomposed into a finite set of triangular regions, connecting the vertices and edges of the polygons. A bridge building approach is used to construct triangular regions. It connects the outer polygon and obstacle polygons so that all lines are connected without any discontinuity. Then the path of the robot is found by connecting the mid-points of the sharing line of the triangles. In other words, the path enters and leaves triangular regions via the mid-points of the common boundaries. Because of this sequence, the path is always parallel to any boundaries in the map and the path is not smoothly connected, in other words robot must turn at the connecting point without changing its position. The optimal path will be the shortest path, but it is not always the minimum time path. It also depends on the number of turns which the robot has to make.

Byung-Kwon Min [8] introduces two methods, Exploration Quadtree and Temporary Goal for the robot exploring unknown environment. Exploration Quadtree provides the information on quality of the regions concerned in a robot's working environments using "entropy of cells" calculated from the probability of the cells. Basically what they do is to set a temporary goal in a certain region by following the rules they introduce. Then when the robot either reaches the goal or finds it is unreachable,

another temporary goal will be set. To determine a temporary goal, a region is divided into four equal sized rectangles. Then these rectangles are divided into even smaller four rectangles until the map achieves a desirable resolution which is determined considering memory size of the on-board computer and size of the cells in a map.

The idea of imaginary forces acting on a robot has been suggested [12]. The Potential Field Method and the Virtual Force Field Method which were first developed by Carnegie Mellon University assume that obstacles exert repulsive forces, while the goals apply attractive forces to the robot. The resultant force vector is computed to be a desired path.

Borenstein [12] presents his Vector Field Histogram (VFH) method with comparison with other techniques such as the edge-detection method, the certainty grid for obstacle representation, and potential field methods for mapping and the virtual force field (VFF) method for navigation. VFH (Vector Field Histogram) method was developed to overcome the shortcomings of the VFF (Virtual Force Field), which are:

1. Robot sometimes would not pass through the door because of the repulsive force from the both side of the doorway.
2. When the robot travels a narrow corridors, it is okay as long as it stays at the center. But once it is off the center, the robot oscillates because of the strong repulsive forces from the closer wall.

To remedy these shortcomings, the VFH method uses a two-stage data-reduction technique rather than a single-step reduction used by VFF method. A one-dimensional polar histogram around robot's momentary location is used in VFH method.

The on-line navigation scheme constructs a surrounding local map while traveling, then integrates it into a global map. An equivalent approach then can be used to navigate in the surrounding environment to achieve a local goal [17].

In the navigation problem, Kalman filters are also commonly used to combine and filter out the noisy sensor data and predict signals in communication and control problems [20], [21], [22]. The navigation problem is usually highly non-linear in practical sense, thus it must be treated by "extended" or non-linear Kalman filter approach. The main advantage using the Kalman filter is that it is recursive, eliminating the necessity for storing large amounts of data. It requires, however, good initial conditions and the noise must obey white Gaussian distribution for dead reckoning. In the case of tracked mobile robot and ultrasonic sensors, neither of these conditions is met.

Chapter 3

The Testbed Mobile Robot ANT (Autonomous Navigation Testbed)

The mobile robot platform which was employed for the performance test of the proposed algorithm is the Autonomous Navigation Tested (ANT) shown in Figure 3-1. ANT was first developed in 1993-94 academic year as an undergraduate senior project, and had been modified to compete in a robot contest. Because ANT is originally designed for outdoor application, it is a tracked mobile robot powered by four 85 Amp-hour, 12 Volt batteries. It is propelled by two 1.5 HP DC motors. The robot contains seven Panasonic Ultrasonic Proximity Transducers around the perimeter of the body, and an optical encoder on each track. The central processor is a Pentium 100 MHz IBM-compatible PC, and Microsoft Windows 95 is installed as an operating system.

Figure 3-1: The Mobile Robot Testbed ANT.

3.1 Robot Geometry

The plane view geometry of ANT is shown in Figure 3-2. Each track is driven by one of the motors and controlled by PWM (Pulse With Modulation) motor amplifiers which are connected to the central processor through a RTD ADA2110 digital/analog interface board. Seven of the ultrasonic range sensors are mounted on the bumper surrounding the robot, two on each side and three in front. All the ultrasonic sensors are connected to the DMI (Distance Measuring Instrument) via a multiplexer. The DMI is also connected to the RTD ADA2110 A/D board. The origin of the robot's two-dimensional coordinate frame is taken to be on the center between the two tracks and 32 inches from the front sensor, so that the robot's equation of motion can be simplified because the center point can be assumed the center when the robot is turning.

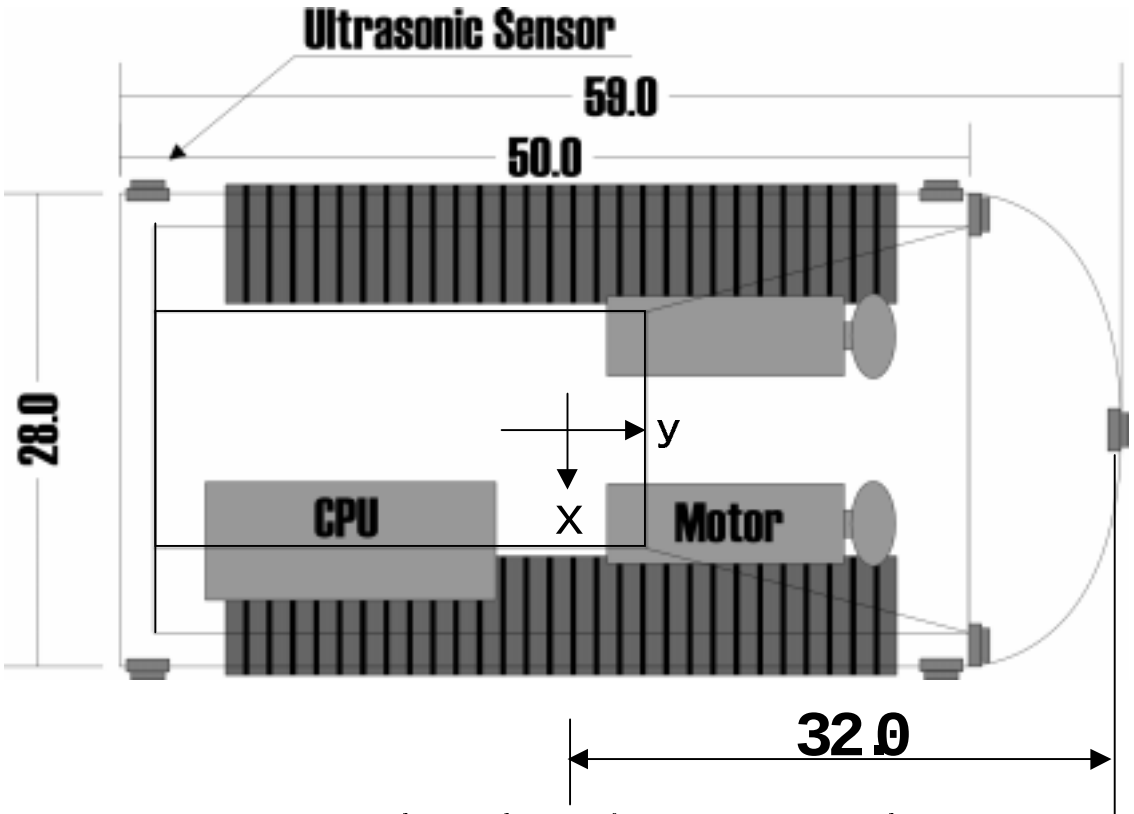


Figure 3-2: Schematic drawing of ANT (Dimensions in inch)

3.2 Motor Control

The robot's motors are powered by four 12 volt batteries mounted between the two tracks, and controlled by the PWM motor amplifiers. The basic function of the Pulse With Modulation is that it sends a voltage pulse instead of a continuous voltage signal. For example, if 50% output is requested, the PWM sends pulses of maximum voltage in a square wave signal with 50% duty cycle. Thus, the total output is averaged to be 50% of the maximum output. The advantage of this technique is that it reduces the power consumption of the batteries. The motor amplifiers have PWM built-in, and tachometer feedback for more accurate operation.

The central processor sends signals to the motor amplifiers through the RTD ADA2110 interface board's Digital-to-Analog conversion ports as shown in Figure 3-3. The motors produce full speed forward which is 16.25 inches per second at receiving 10 volts, full speed backward which is also 16.25 inches per second at –10 volts, and stop at 0 volts. The ADA2110 board is able to convert a 12 bit digital signal to analog –10 to 10 volts which corresponds to 0 and 4095 in decimal and 000 and FFF in hexadecimal. Thus the motor control procedure is to first divide the number into two parts, a least significant byte and most significant byte, then send it to each port. Each motor is equipped with two optical encoders for feedback. The encoders are read by two HTCL chips which are connected to the central processor through Analog Devices DT2817 digital-to-digital interface board.

The tracks have only one degree of freedom, back and forth. Because of this characteristic, the tracks must slip for the robot to turn. One of the major advantages of the tracked vehicle in maneuverability is that it can turn on its axis. This means that the

minimum turning radius is greatly reduced to the length from vehicle's center of turning to the outmost point of its perimeter. Thus tracked vehicles are suitable to maneuver in narrow environments which require sharp turns. However due to the fact that tracks must slip to turn, the prediction of the turning curves highly depends on the operating surface condition.

3.3 Ultrasonic Sensor System

The ultrasonic sensor system consists of seven Polaroid transducer sensors to send and receive ultrasonic signals, one Distance Measuring Instrument (DMI) to convert the transducer's data into an actual distance and to generate a corresponding analog voltage, and one seven-channel multiplexer to allow a maximum of seven transducers to be used with one DMI. The seven sensors are mounted on the bumper of the robot, two on each side and three on front as shown in Figure 3-2.

The main advantage of the ultrasonic sensors is that they are inexpensive. The disadvantages to them are their lower directionality and lower accuracy when the object is not facing perpendicular to the sensor. Although the ultrasonic sensors can fire and receive the echo at a rate of 10 milliseconds, the multiplexer requires 75-80 milliseconds in duration to change the channels according to the manufacturer's specification. The analysis and calibration of the ultrasonic sensors are presented in the Chapter 4.

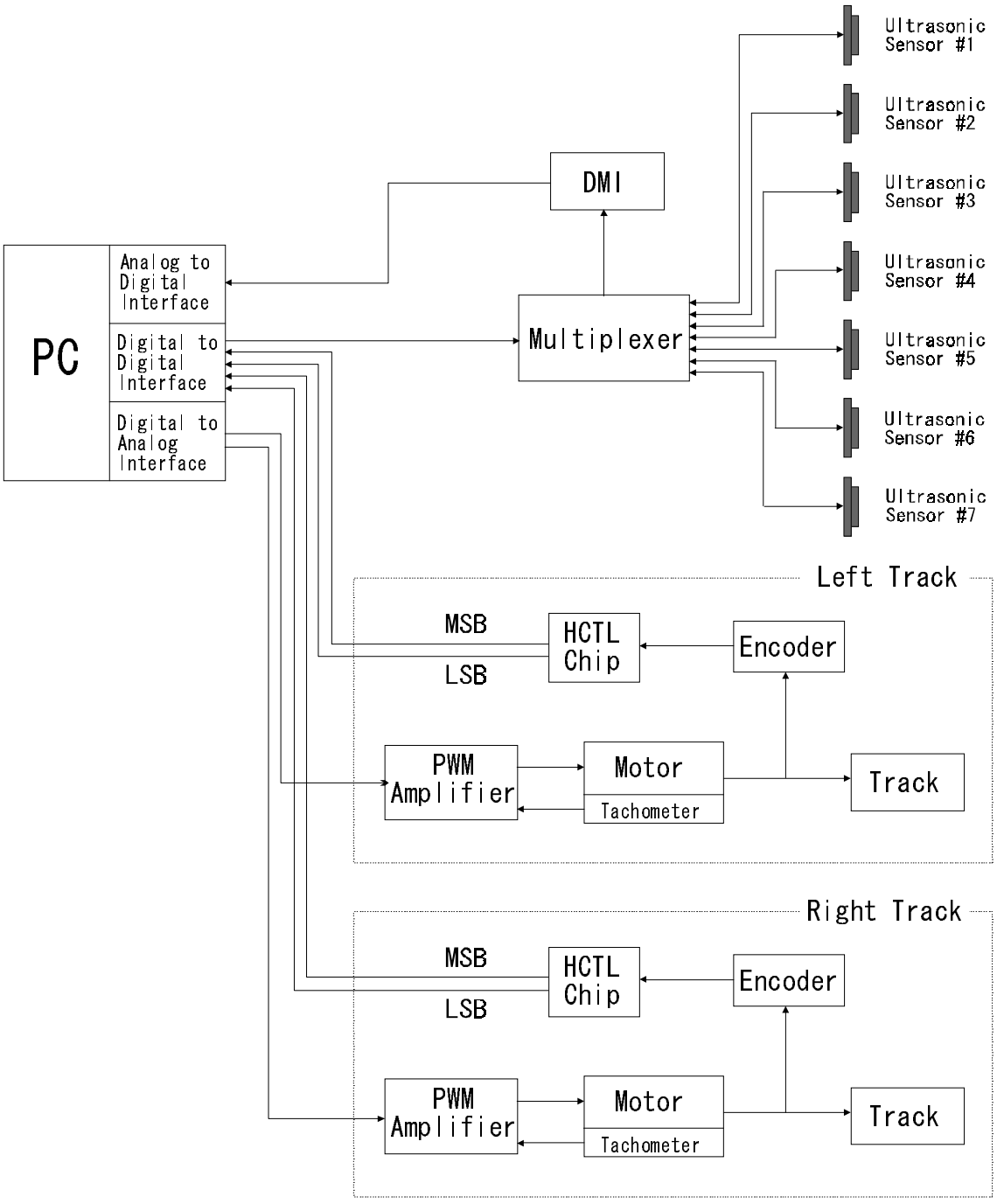


Figure 3-3: Schematic diagram of ANT's control system

3.4 Robot Hardware Control System

All the sensors are connected to the on-board Pentium 100 MHz PC-AT compatible computer using two data acquisition boards. The computer and all the sensors are powered by two 12 volts batteries which are independent of the motor batteries. The programs are written using Microsoft Visual C++ 5.0 and the four source codes are shown in Appendix C.

Chapter 4

Technical Approach

This chapter presents the strategies and algorithms to accomplish the objectives described in the section 1.2. The methods to accomplish the task can be categorized into five parts, which are ultrasonic sensors, mapping, feature recognition, map matching, and navigation.

4.1 Optical Encoders

The shaft of the each motor was painted half black as shown in Figure 4-1, and has two optical sensors, spaced 90 degrees apart. Each sensor consists of an LED and a photo-transistor. Light emitted by the LED is reflected to the photo-transistor by the unpainted half the motor shaft, but absorbed by the black painted half. The resulting square wave signal from each sensor are sent to a Hewlett Packard HCTL2016 decoder chip. The quadrature method was used to determine the direction of the shaft rotation. The HTCL2016 chip includes a quadrature decoder, 4x resolution enhancement, and a 12 bit up/down counter. An output buffer is used to prevent

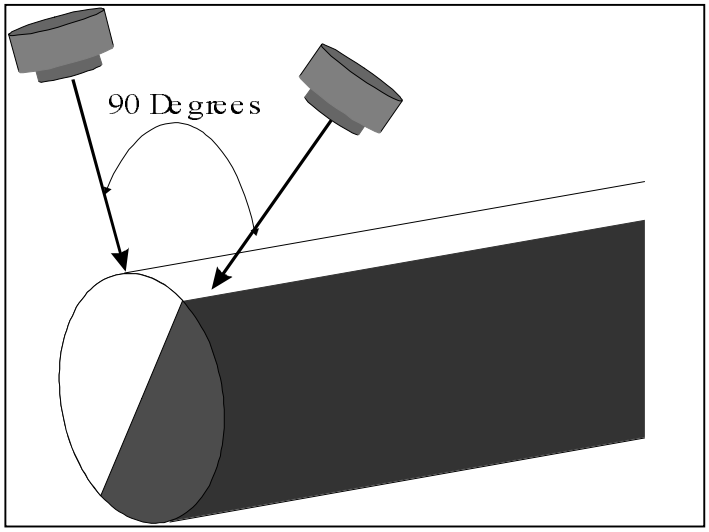


Figure 4-1: The motor shaft is painted half black and two optical encoders were placed in 90 degree.

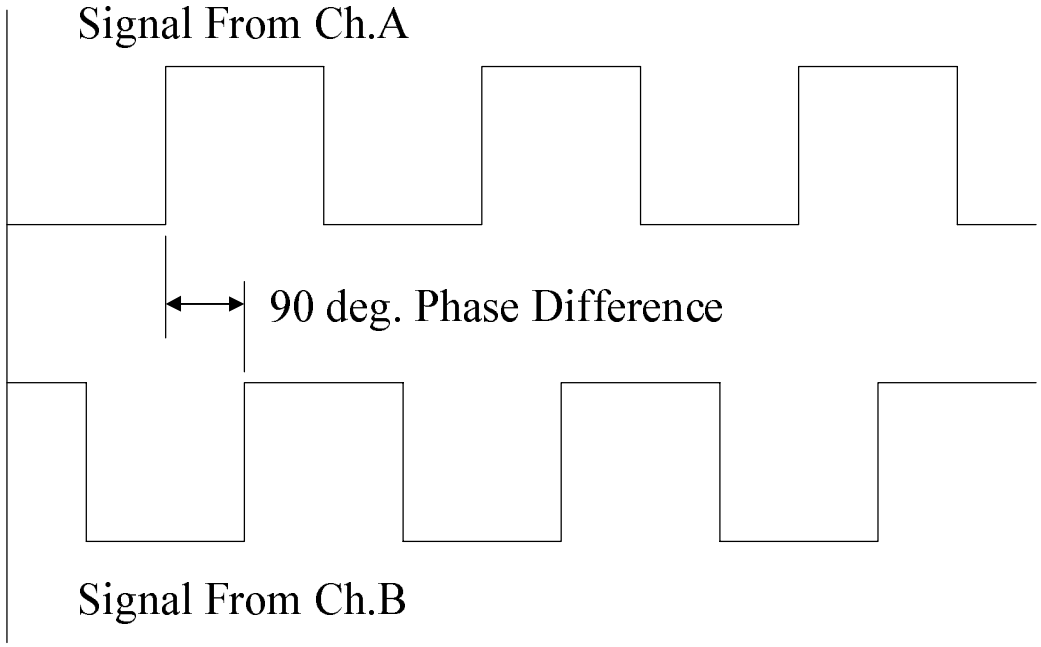


Figure 4-2: Two optical encoders return signals which have a 90 degree phase difference. From the phase difference, the counter chip can determine the direction of the rotation of the shaft.

loss of counts when the chip is being read.

Because each port of the digital-to-digital interface board can provide 8 bits of data and the HTCL counter chips return readings in 12 bits, the two ports are connected to one HTCL chip, one is for the least significant byte (LSB) in 8 bits and the other is for most significant byte (MSB) in 4 bits. The two ports are combined and converted to distance. The conversion factor is 90 counts for 1 foot. So the converting equation becomes:

$$\frac{(MSB \times 256 + LSB) \times 12}{90} - (\text{Last Reading})$$

which gives the distance traveled in inches since last reading for each track. The software includes logic to detect when counter roll over.

From the distance traveled on each track, the trajectory of the robot can be calculated. Although Tenney [20] experimentally determined the relationship between the

turning radius and the ratio of inner and outer tread speed for ANT, it is highly dependent on the friction forces on the track, and its accuracy varies greatly depending on local floor conditions. The turning angle can be calculated from the geometry of ANT by assuming that the center of the turn and the geometrical center of the two tracks match. The equation becomes

$$\text{Turned Angle} = \frac{\text{Distance Traveled by one track}}{\frac{1}{2}(\text{Distance between tracks})}$$

4.2 Ultrasonic Sensors

The Panasonic Proximity Ultrasonic Transducer works as both diaphragm transmitter and receiver. The Distance Measuring Instrument (DMI) sends a 300 volt signal to the transducer to vibrate the diaphragm to generate the ultrasonic waves. The typical ultrasonic wave consists of four different frequencies in a range from 48kHz to 56kHz. By having different frequencies, the transducer can avoid blind zones which may be caused by standing waves in which a transmitted ultrasonic wave interferes with the reflection wave.

The reflected wave impinges on the same diaphragm, which now acts like a microphone. The return signal is amplified by a gain-controlled amplifier in which the gain is a function of time. Reflected signals from farther away are amplified more, since their signal strength is weaker. The DMI converts the detected distance by a sensor to a voltage signal ranging from 0 to 10 volts. Then the computer receives the digital signal through the RTD ADA2110 analog-to-digital/digital-to-analog interfacing board.

Seven ultrasonic sensors were used on the robot to measure the distances to walls and obstacles. Since the DMI, the multiplexer, and the power supply were already installed on the robot, calibration was done using the robot and its onboard computer.

The primary task of the ultrasonic sensors is to detect the walls of the hallway to build local maps. Therefore, for the calibration, the actual wall in the operating environment was used as the object to be detected by the sensors. Because the sensor fires an ultrasonic wave and measures the time till the reflection comes back, it may not be able to detect the wall if it is not nearly perpendicular to the transducer.

First the voltage signals from the DMI were sampled to calculate the distance in inches by comparing the actually measured distance with a tape measure. The readings

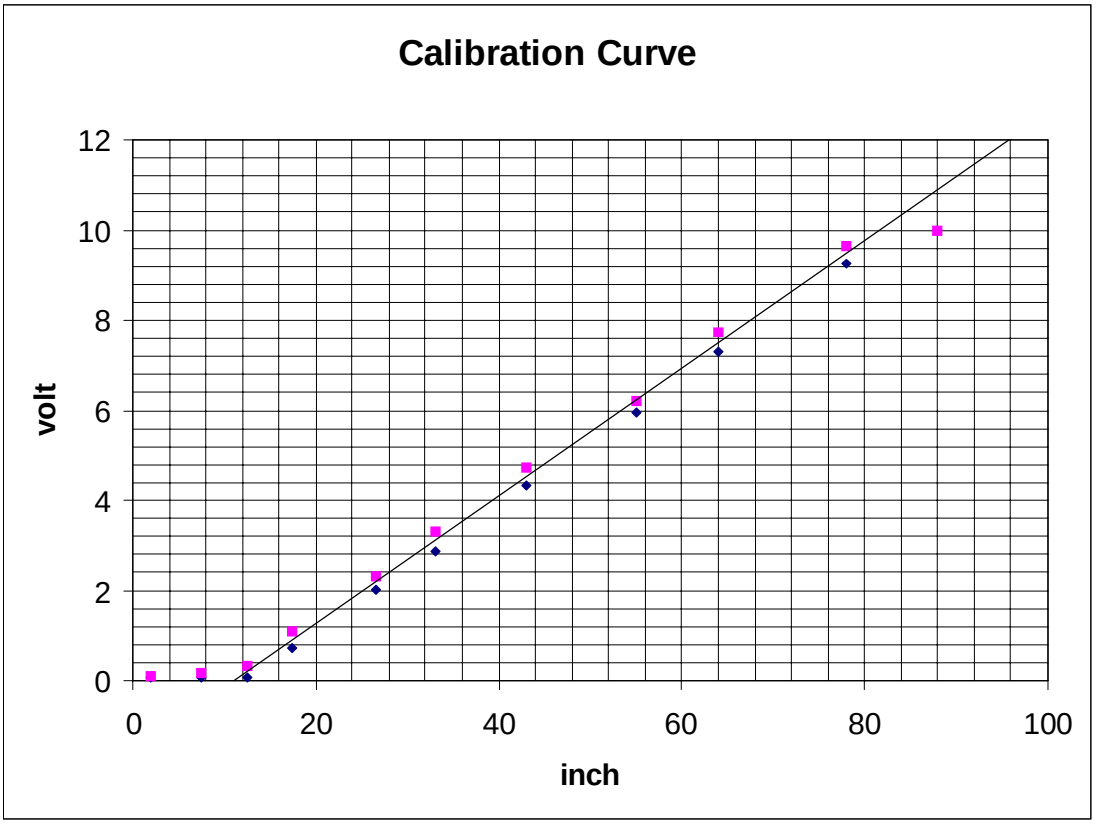


Figure 4-3: Ultrasonic sensor calibration Curve

are taken directly from the display on the DMI to avoid the noise caused by the multiplexer and the A/D board. The maximum and the minimum numbers of the DMI readings were taken at each position by observing the DMI display. Because it can be said that the signal fluctuation is much larger than the individual errors, two sensors mounted on the front were used to derive the calibration formula. The plot is shown in Figure 4-3 and a linear curve fit was applied to the plot and it is expressed as

$$\text{Distance(in.)} = \frac{\text{volt} + 1.567}{0.1417}$$

From this equation and the 0-10 volt specification, the theoretical minimum and maximum distance that the sensor can detect are obtained as 11.06 inches and 81.63 inches respectively.

The sensors were mounted on the robot bumper to detect walls near the robot. Three sensors were mounted on the front and two were placed on each side to aim at the same walls for better accuracy. None of the sensors were placed on the back so that an operator can follow the robot for experimental purposes without providing an unexpected obstacle to the navigation program. The actual location and the firing sequence which is corresponding to the sensor port number are shown in Figure 4-4.

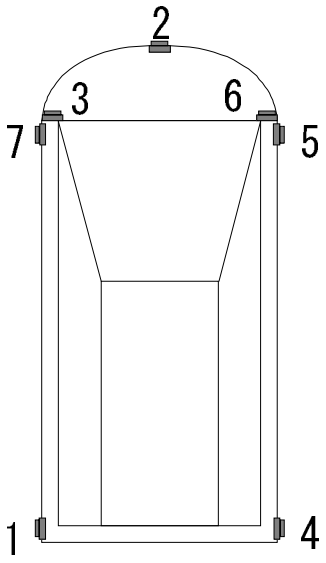


Figure 4-4: Ultrasonic sensor locations and firing sequence

4.3 Mapping Technique

The basic concept of the mapping technique is to combine the ultrasonic sensor data and dead reckoning information together. The optical encoders on each track can provide an idea how much the robot traveled since the last reading. Although it is not precise enough, it can also give a change in orientation from the difference between readings from each track. According to the dead reckoning data, the program can locate the robot in a map, and the ultrasonic sensor information can be transformed from the robot coordinate system to the global coordinate system to build a sensor map.

The Histogramic In-Motion Mapping, which was developed by Borenstein, et al. [11], [12], was applied in this project, because it has been shown to be computationally fast and was effective in experiments performed by Borenstein and his associates. The idea of the Histogram map is a two-dimensional Cartesian grid in which each cell has a certainty value to indicate the existence of an obstacle in the environment. When an ultrasonic sensor reads the distance to an object, it cannot determine the bearing angle to the object because, as shown in Figure 4-5, the sensor has a conical field of view which is approximately 16 degrees. Although these sensors are assumed to have 30 degrees beam width in the

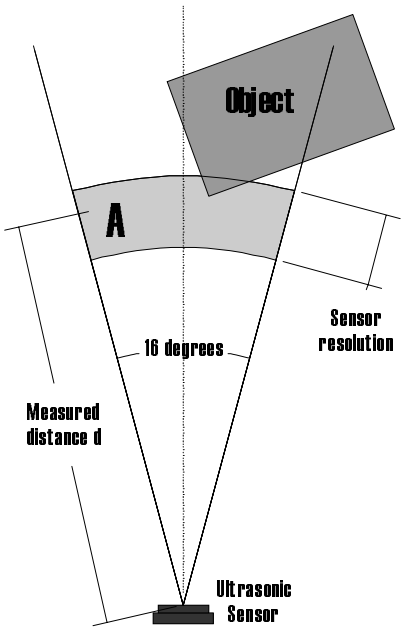


Figure 4-5: Two-dimensional projection of the conical field of an ultrasonic sensor. The dotted line represents the acoustic line and a range reading d indicates the existence of an object somewhere in the shaded region A.

literature, the multiplexer used in the ANT appears to attenuate the return signals somewhat, making the effective beam width for ANT's sensors rather narrower, particularly for returns from smooth walls.

Each cell has a certainty value between zero to eight, with zero for vacancy and higher certainty values indicating higher possibilities of existence of an obstacle. As suggested by Borenstein, the certainty value would be added to only one cell which is along the acoustic axis of the sensor. This method may appear an over simplification because the actual position of the obstacle and the cell on the acoustic line may differ significantly as indicated in Figure 4-6(a). It is possible that an error between the actual and estimated position of an obstacle could become a serious problem if the vehicle is still. Thus Borenstein's method requires the vehicle to keep moving continuously so that the histogram map is updating continuously. A cell and its neighboring cells are repeatedly incremented by combining the ultrasonic data with the dead reckoning, as shown in Figure 4-6(b).

Although this method has tendency to make every obstacle look like a wall, the most important consideration was simply to detect the obstacle and avoid it. In this work, it is not necessary to distinguish a chair from a waste basket. The advantage of this approach is that the readings from a sensor will be confirmed by other sensor readings. For example, in this project, the robot has two sensors on each side. As the robot travels, the certainty values built by front sensors of each side will be increased by confirming by the rear sensors. This requirement is most suitable for a robot that can make smooth curves, although the ANT is programmed only to move straight ahead and turn on its axis.

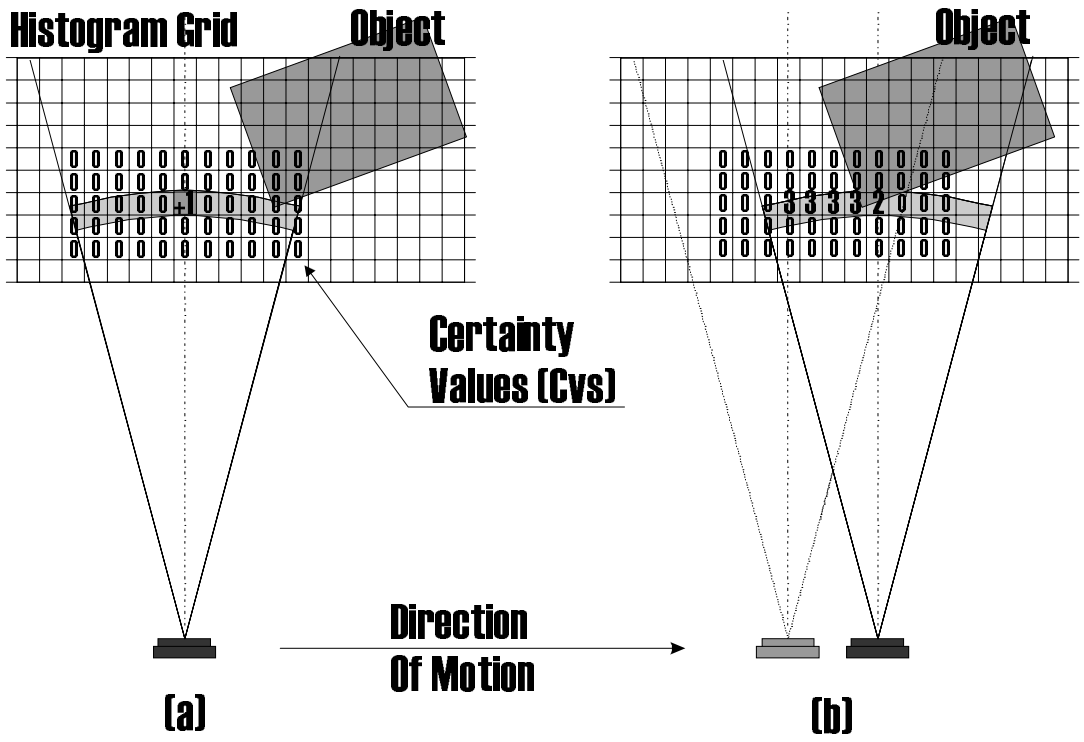


Figure 4-6: (a) Only one cell is incremented for each range reading. With ultrasonic sensors, this is the cell that lies on the acoustic axis and corresponds to the measured distance. (b) A histogramic probability distribution is obtained by continuous and rapid sampling of the sensors while the vehicle is moving.

Borenstein also presents a noise reduction method by subtracting the certainty values. The histogram map becomes more reliable by not only adding a certainty value to the cell on the acoustic axis which corresponds to the measured distance, but also by subtracting some value from the cells on the acoustic axis, which are between the measured distance and the sensor position. This method seems to reduce noise in the histogram map in Borenstein's application where his robot has a ring of 24 ultrasonic sensors placed with 15 degree interval, each connected to an individual DMI.

However, this noise reduction method is not applied to this project because fewer sensors were used and all of them are connected to single DMI using the multiplexer which provides slower reading intervals, inaccuracy of the sensor data, and the narrower effective range as discussed in the earlier section. The ultrasonic sensors also often give false returns. Moreover, the false readings may delete the correct readings on the histogram map if the noise reduction method is applied. Additionally, many of Borenstein's

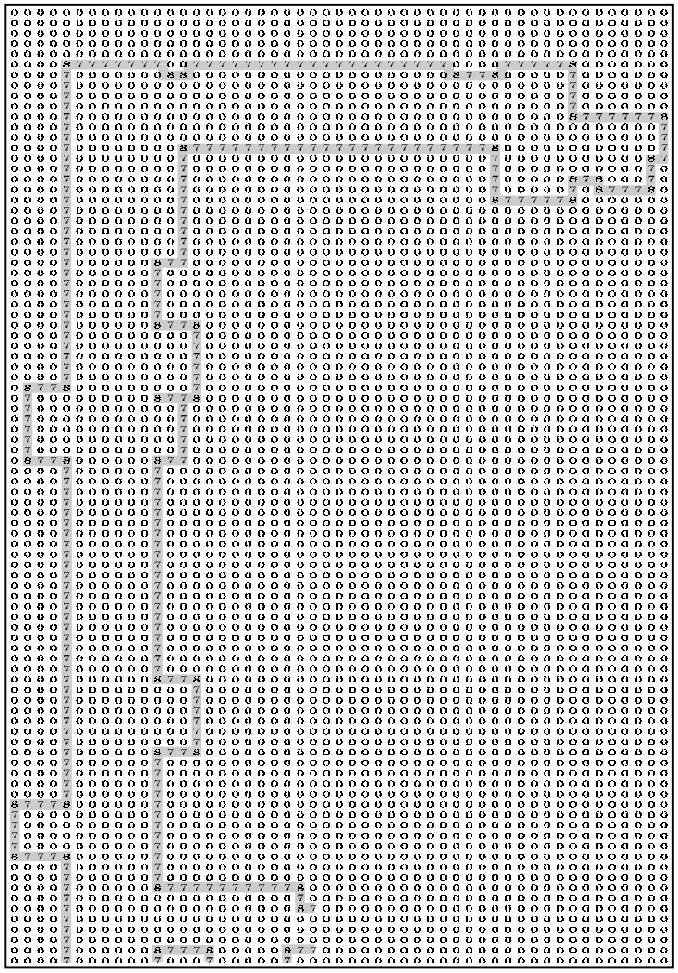


Figure 4-7: The environmental map in a matrix format given to the program in the beginning of the procedure

experiments were performed with plastic pipes as the obstacles. Although they are thin, they present a vertical cylinder as the reflecting surface, which is an ideal omnidirectional reflector. The sensors sometimes give false readings to an object, especially one which is not perpendicular to the acoustic line of the sensor. Thus, his obstacles, which were cylindrical pipes, were more acoustically reflective than the walls in this project, because those pipes can always reflect the ultrasonic wave straight back to the sensor.

To perform a map matching, two maps were prepared. One is the model map

which describe the robot's operating environment as shown in Figure 4-7. It was loaded into the program at the beginning of the procedure. The other is the sensor map, which is built by the sensor readings and the dead reckoning combined together. The maps are constructed as a form of a matrix in which each cell represents a one-foot by one-foot square region. Both maps are the same size, a 92 by 52 matrix which represents an area of 92 feet by 52 feet in this experiment. The sensor map is set to zero, and the model map is loaded into the navigation program in the beginning of the algorithm. As shown in Figure 4-7, the model map uses three numbers, 7 indicates the wall, 8 for corners, and 0 for vacancy. For the sensor map, the sensor readings are based on the distance of the obstacle from the ultrasonic transducers on the robot. Once a reading is taken, the coordinates of the reading are transformed from sensor coordinates first to loacl coordinates and then to global coordinates. The local coordinates systems are set at the center of the robot as shown in Figure 4-8. The transformation is done by using transformation matrix;

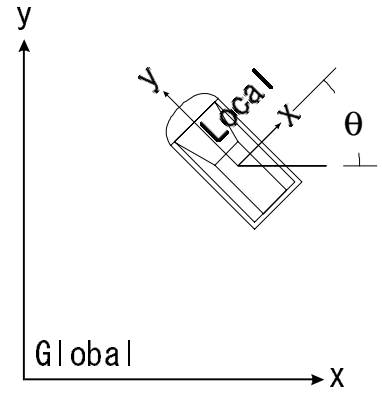


Figure 4-8: The global coordinate frame and the local coordinate frame attached to the robot

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}_{global}^{sensor} = \begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix}_{global}^{local} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}_{local}^{sensor}$$

where the matrix in the right-hand side is the position of the local coordinate frame with respect to the global coordinate system and the column vector in the right-hand side is the obstacle location with respect to the local coordinate system.

4.4 Feature Recognition and Map Matching

The first step of the feature recognition is to filter out noise and group the matrix entries. Since the range of the sensors is only about 7 feet, the program extracts local maps which are 9 cells by 9 cells or 9 feet by 9 feet, surrounding the robot from the global maps to perform feature recognition as shown in Figure 4-9. The two extracted local maps are named the local model map and the local sensor map whereas the original maps sized 92 by 52 are named the global model map and the global sensor map. The local sensor map may be translated and rotated with respect to the local model map, as a results of errors in the dead reckoning computation. The map matching algorithm attempts to match features in the two maps in order to detect and correct the dead reckoning errors.

Cells with the certainty value of 1 are considered as noise and are filtered out because the sensors give false readings once in a while as discussed earlier in this chapter. This error may be caused by undetermined defects in the hardware, probably the multiplexer. Thus the certainty value of 1 can be filtered out as insurance. To filter out the certainty value of 2 makes the sensor map less accurate because the cells that are detected by only the front sensors on each side have fewer certainty values when the rear sensors on

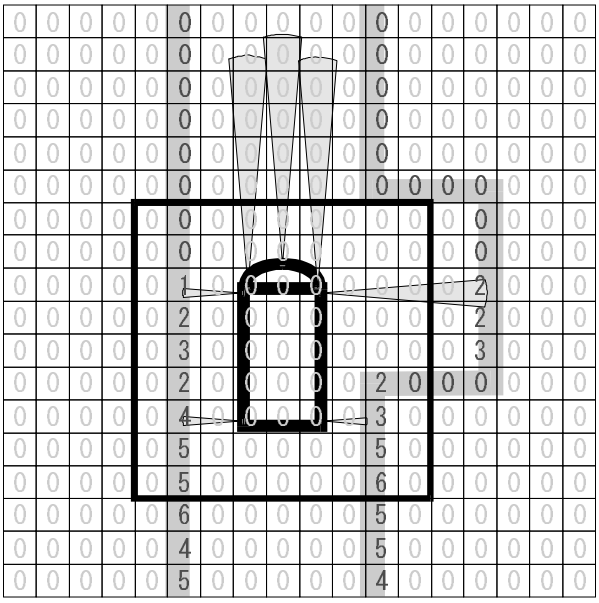


Figure 4-9: An example of mapping when ANT is traveling the hall way. The bolded line represents the region used by the feature recognition procedure.

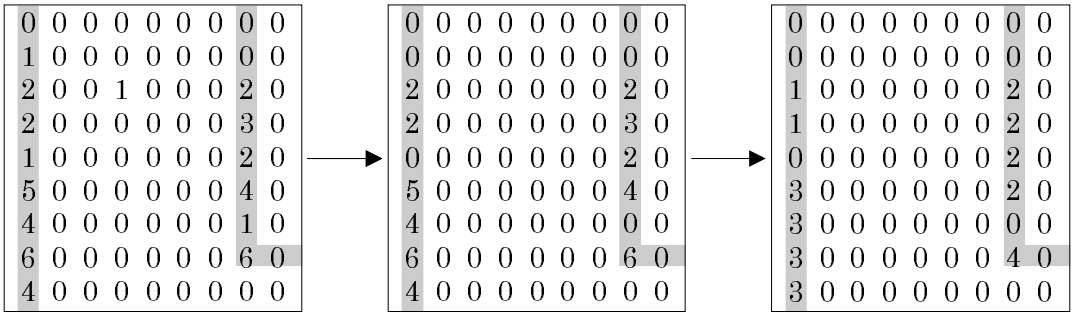


Figure 4-10: (a) Raw map data. (b) Filtered map data. (c) Cells are grouped and numbered.

each side are not reached yet as illustrated in Figure 4-9. The square box with bold lines in the middle of Figure 4-9 is the region where the feature recognition will be performed. The sensor readings are sampled slowly enough to eliminate cross talk error, although the exact number of pulses per second can not be determined because they are not fired periodically, i.e. other subroutines are executed between each reading.

The program searches for the continuities between groups of cells and numbers them as shown in Figure 4-10 for example. The procedure starts with raw data; Figure 4-10(a). The center of the robot is located at the center of the matrix. The data is filtered to eliminate the certainty value of 1 to become filtered data; Figure 4-10(b). Next, the program checks each cell and its neighboring cells for continuity. Each group of cells is numbered and the certainty values are replaced with the numbers which indicate the groups as shown in Figure 4-10(c). The basic flow chart is shown in Figure 4-11. Finally, the map with grouped cells is sent to the feature recognition algorithm. This recognition algorithm consists of two parts which are wall finder and corner finder. The program checks for walls and corners to estimate the location of the robot compared with the model map.

4.4.1 Finding Walls

In the wall recognition algorithm, groups of cells are divided into three categories; long wall, short wall, and obstacle. The definitions of these categories are:

- Long Wall: a group of cells which has three or more in-line cells or four or more cells which are lined in same direction.
- Short Wall: a group of cells which has more than one and less than three in-line cells.
- Obstacle: a cell which does not have any neighboring cells.

First, the algorithm starts searching for the features listed above from the center of the local sensor map outward. Thus the closer walls to ANT will be detected first.

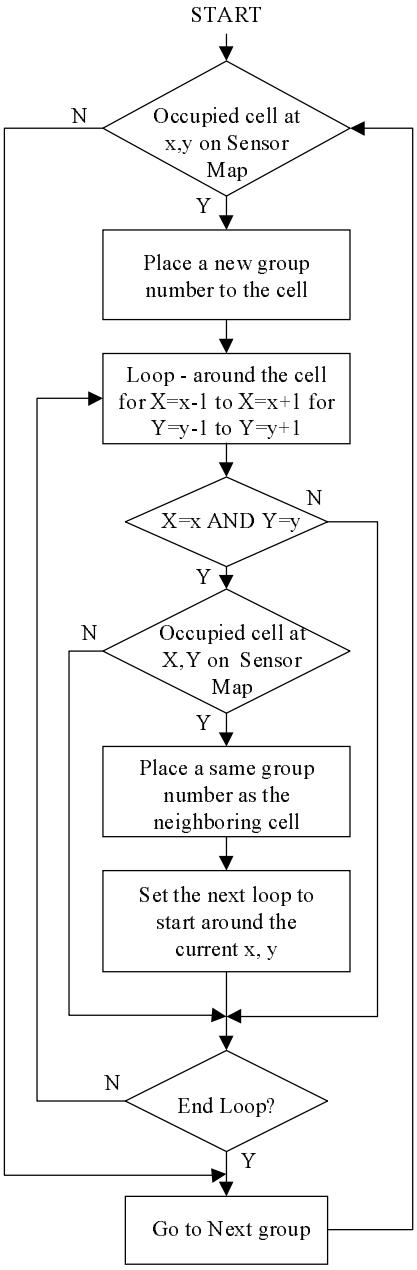


Figure 4-11: The basic flow chart of the obstacle grouping algorithm.

As the next step, the information on the location of these groups is handed to the map matching algorithm. In the map matching algorithm, a 9 by 9 matrix of a local model map is extracted from the given global model map at the same location where the local sensor map is expected to be. Thus, the location of the walls on both local maps should match if the

robot location is correctly computed from dead reckoning. To compare the locations of the walls between the model and sensor map, the location of the walls in the model map are compared according to the information from the sensor map case. Then the positions of the walls are computed from their average positions, i.e., the center of a wall is taken as the location of the wall. The flow chart of the algorithm is shown in Figure 4-12.

Once the location of a wall in the sensor map is calculated, the map matching algorithm searches for a wall in the same direction in the model map. In case both walls don't match, the program calculates the distance between the two along the axis which is perpendicular to the wall as shown in the flow chart. It is not unusual for the distances thus calculated to be in

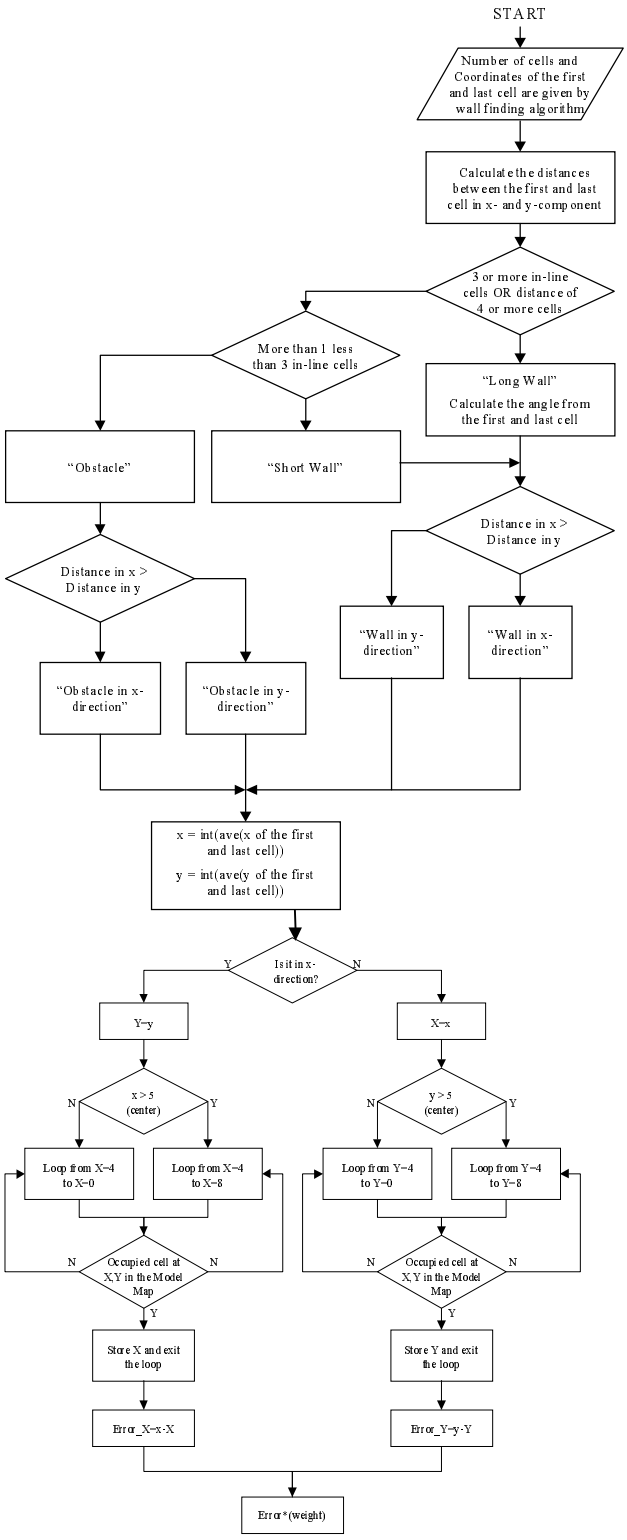


Figure 4-12: The basic flow chart of the wall matching algorithm.

conflict with each other. To account for this, the correction parameters are weighted, with a weight of three for a long wall, two for a short wall, and one for an obstacle, since the longer walls are more reliably detected than shorter walls or obstacles.

The long walls are also used to determine the actual orientation of the robot. The walls in the operating environment lie along either the x- or y-axis. The angle of a wall is calculated using the first cell and the last cell of the string defining the wall. In a case in which two or more walls are detected, the average of them is taken as the orientation at that interval.

4.4.2 Finding Corners

The corners are the other feature used in the map matching algorithm. Unlike the wall finding procedure, the corner-finding procedure starts by analyzing the model map, because it is almost impossible to find corners from the sensor map due to the difficulty of detecting both sides of the corner in this limited sensor arrangement. As shown in Figure 4-7, the corners are labeled as eight in the model. The local model map is superimposed on the sensor map to check if the cells with 8 have corner features in the local sensor map. The arrangements of the cells which can be identified as corners are:

- A cell with two neighboring cells in perpendicular, Figure 4-13(a).
- A discontinuity of the in-line cells, Figure 4-13(b).
- A cell without neighboring cells, Figure 4-13(c).

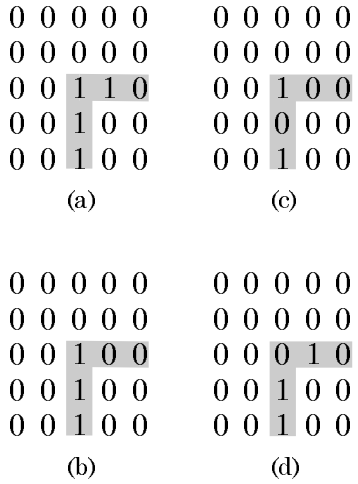


Figure 4-13: Features of a corner

- A vacant cell with two neighboring cells in perpendicular, Figure 4-13(d).

The case of (a) has an obvious feature to be recognized as a corner although it is difficult to detect the walls which are perpendicular to the robot. In such situations, corners most likely appear as in cases of (b) and (c) which have corresponding discontinuities. Even the case (c) is considered as a cell with discontinuities. Although the case (d) does not have a corresponding cell, the two neighboring cells in perpendicular position suggest the existence of a corner.

If a corresponding corner cannot be found in the local sensor map at the point where the corner cells are located in the model map The program then searches the neighboring eight cells for a corner. When the match for the corner is found, the errors are summed and averaged in both axes.

The results from both the wall and corner matching algorithm are then combined to adjust the location of the robot in the model map. If the locations of the model and sensor map don't match, the local sensor map must be shifted according to the errors from the map matching algorithm. The local sensor map matrix is shifted vertically and horizontally as necessary, then stored back to the global sensor map. The orientation of the local sensor map calculated from long walls is used to correct the robot's orientation, rather than to adjust the sensor map. If no long walls are present in the environment, or if none are found in the local sensor map, the dead reckoning estimate of the robot's orientation is used for the next segment.

4.5 Navigation

The ANT has an off-line navigation algorithm which means that the environmental map is given to the robot and path planning is done in advance of the actual navigation procedure. In addition to the environmental map shown in Figure 4-7, the initial position and orientation of the robot and cornering point coordinates at which the robot must make a turn are given. The desired path is constructed by the path planning algorithm by connecting the turning point coordinates as shown in Figure 4-14.

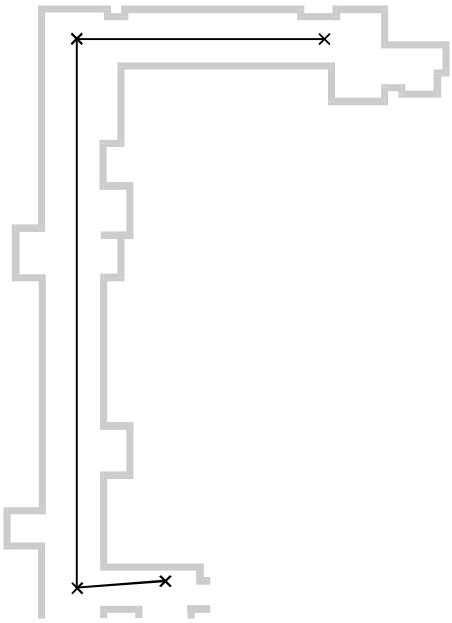


Figure 4-14: The cross signs indicates the cornering points given. The line connecting them is the desired path.

In order to build the histogram, the robot must keep moving. In other words, the initial position and orientation of the robot must be given so that the robot can start the first move. The initial position and orientation need not be too precise, but they must be accurate enough to complete the first interval. Position correction will be done periodically once the robot has traversed the first segment.

The map matching procedure is executed after a local goal has been achieved or when the dead reckoning estimates that the robot has traveled five feet. The robot position error is computed from the results of the map matching algorithm described above. If the next turning point is farther than ten feet, the navigation algorithm sets the next goal ten feet away on the desired path. The robot turns on its axis toward the next goal

and proceeds in a straight line.

After the corrections are made, the robot makes a turn to the next goal. Placing the local goal farther in front of the robot helps to reduce overshoot and makes a generally smoother trajectory. Then the robot travels next five feet or the remaining distance to the next turning point, and the map matching sequences are again executed. The robot stops when it reaches the last point on the given path.

Chapter 5

Experimental Results

This chapter will discuss the experiments which implemented the methods of map building, feature recognition and map matching, and navigation presented in the previous chapter, on ANT. Ten test-runs were performed and then the results of these tests will be discussed.

5.1 Experimental Method and Setup

The purpose of the experiment is to test whether the ANT system meets the objectives defined in the introduction chapter. The test environment is shown in Figure 5-1. As the figure shows, the robot starts at an office room, goes out to the hall way, goes down the hall way and makes a turn, then stops at the end of the hall way. The test environment includes the necessary aspects to test the ANT system, such as long walls and corners. Either side of the wall is in the range of local maps even if the robot is off from the model path. Although there is one spot where both sides of the walls cannot be detected from the robot on the model path, the robot should be able to navigate through by using information obtained at other locations.

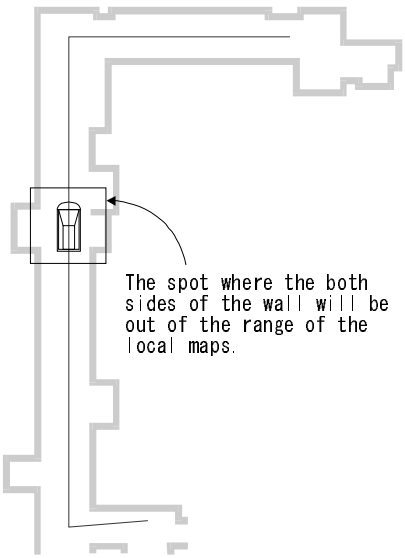


Figure 5-1: The test environmental map and the model path. The square around the robot represents the region for local sensor map.

The navigation algorithm is programmed to output the log of the operations of the feature recognition and map matching to the floppy disc at every map update point so that the algorithms can be checked after the operation. The program also saves the sensor map at the end of procedure as a text file. The actual path is traced by placing post-it notes at the rear end of the robot at every turn. Because the robot only moves straight forward and turns on its axis, the actual path can be measured by connecting the cornering points. Since the position of the rear end of the tracks is traced, not the origin of the robot coordinate, the traced points must be modified to locate the center of the robot according to the geometry. Then the actual path and the model path can be plotted, and also the error between them can be calculated to check the objective 3.

By analyzing the log file and the map file, it can be checked that the system meets the objective 1 and 2. The map building procedure can be checked by the global sensor map which is generated after completing the navigation. The log file includes the coordinates where the ANT executes the map matching algorithm and the local model map and the local sensor map at the corresponding location. The log file includes the record of the map matching and corner finding procedures, such as locating and categorizing the walls, estimating the coordinates of the walls and comparing errors with the walls in the model map. Because every step of the procedure is recorded in this log file, the algorithm can be checked by analyzing the log file. The estimated coordinates can be compared with the actual location traced by the post-it notes.

5.2 Map Building

The map building procedure can be checked from the sensor map which was saved to the disc at each cornering point. Figure 5-2 is the sensor map built at test run #1. It was saved as a form of a matrix. The numbers indicate the certainty value of each cell. The zeros were deleted and the actual map is superimposed on the global map for easier observation. The sensor map was successfully built in this test run. After the filtering out the ones which also appear in the middle of the hallway, the occupied cells follow the walls successfully. However, there are some spots that the wall was not detected. This is because the robot approached too close to the wall, less than its minimum range, so that the ultrasonic sensors were not able to detect the wall. The same situation appeared more obviously in the test run #2 as shown in Figure 5-3. The right wall was not clearly detected as the left wall, where the robot was off to the right from the model path as shown in the

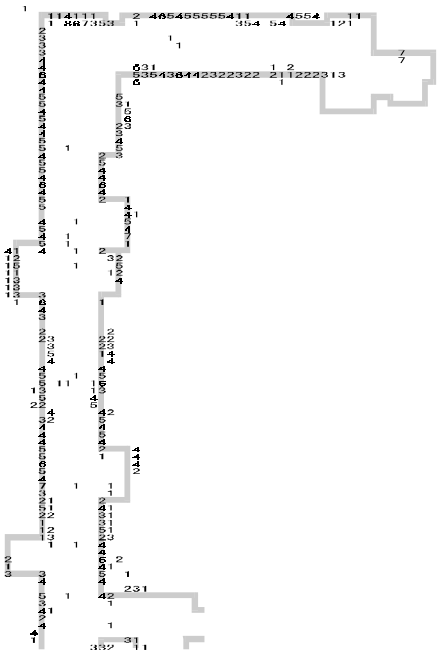


Figure 5-2: The sensor map from test run #1.

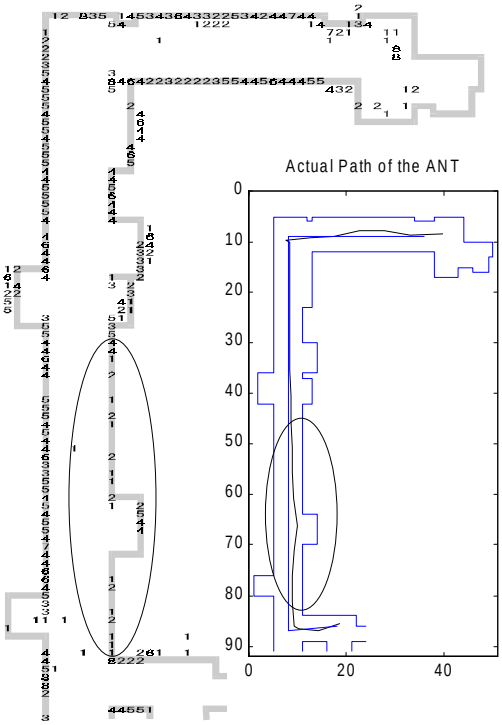


Figure 5-3: The actual path of ANT and the sensor map built in the test run #2. The circled area indicates the spot where the wall is hardly detected.

circled area in Figure 5-3.

It can be said in general that once the robot starts following the model path and traveling parallel to the wall, the sensor map becomes more accurate and the walls are detected more reliably as straight lines. By comparing the results from test run #5 and #8 in Figure 5-4, the sensor maps came out differently. Although the test run #5 was a failure case, the sensor map was built successfully until the program was terminated. The robot followed the model path with an error of less than six inches in the circled segment. On the other hand, in the test run #8, the error between the model and actual path was approximately ten inches. Although ANT followed the model path as well as the test run #5, two sensor maps differ significantly as Figure 5-4 shows. The important difference between two can be found in the error plots shown in Figure 5-5. ANT followed the model path more smoothly in the test run #5, while it

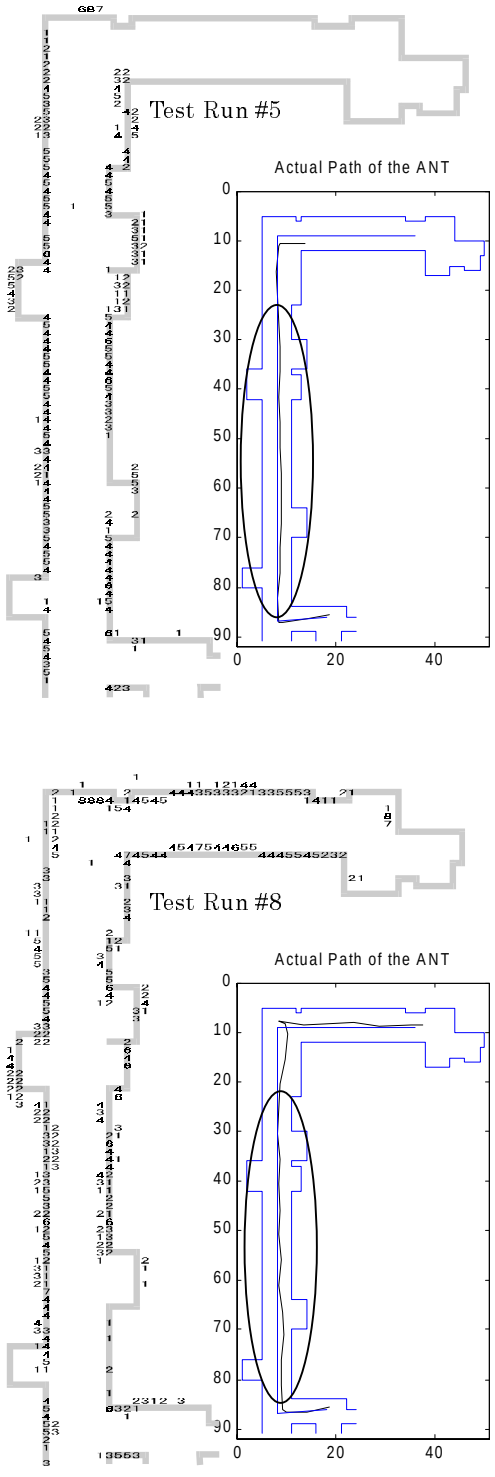


Figure 5-4: Resulting sensor maps and the actual paths of ANT from test run #5 (top) and test run #8 (bottom)

was rather oscillating in the test run #8. In the Figure 5-5, the two solid lines indicate the turning points and the segment between two dotted lines corresponds to the circled segment in Figure 5-4. It is concluded that the sensor map becomes more reliable when the robot is parallel to the walls. It is either because of the ultrasonic sensor's characteristics or because the resolution of the map is not fine enough to build an accurate map when the robot is angled. In other words, when the coordinates of the walls are transformed from the robot coordinate system to the global coordinate system, the standard rounding is applied to add a certainty value to the corresponding cell. This procedure may cause the roughness in the sensor map.

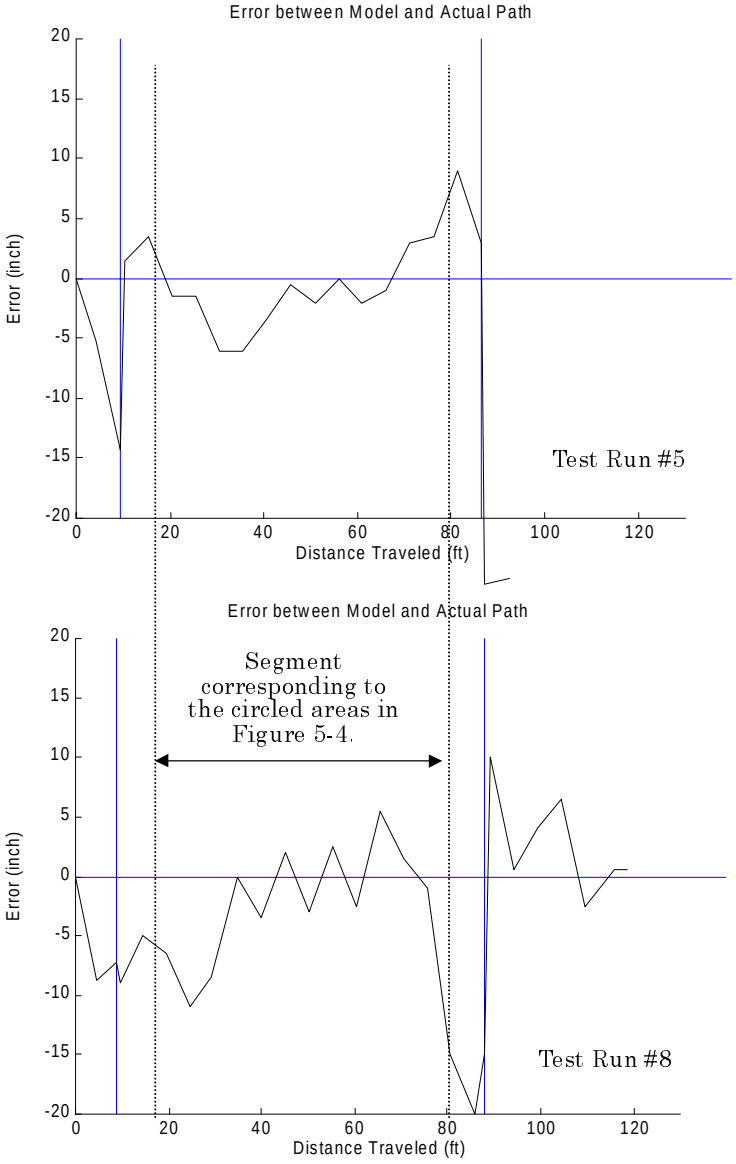


Figure 5-5: Plots of Error between the model and actual path from test run #5 (top) and test run #8 (bottom)

The map building procedure performs

successfully even in the case where navigation was failed and ANT had to be stopped manually. The failure cases are discussed later in this chapter. All of the results obtained from the ten test runs are presented in Appendix B.

5.3 Feature Recognition and Map Matching

The log file includes all the procedures of the feature recognition and map matching. It was used to check the feature recognition and map matching algorithm after the navigation. Figure 5-6 shows an example of map matching procedure taken from the log file. The upper matrix shows the local sensor map and lower maps are showing how the robot position in the model map is corrected according to the information obtained from the sensor map. In the sensor map, numbers represent the groups of cells. On the other hand, in the model map, 7 represents a wall, 8 for a corner, 2 for the model path, and 9 for the estimated robot path. The center of the robot is located at the center of the map. According to the log file, the map matching procedures were performed as follows in this sample.

- In the sensor map, group 1 which consists of only one cell is identified as a part of a wall which is supposed to be one cell to the left as compared with the model map below. The weight of group 1 is low because it contains only one cell.
- Group 2 is identified as a long wall in the y-direction, because it has six in-line cells. As superimposed on the model map, the location must be adjusted one cell to the right. The angle of the wall is also computed to be 0 degrees, which means ANT was traveling parallel to the wall.

- Although group 3 contains four cells forming a corner, the feature recognition algorithm can only determine it as a short wall in the y-direction with possible discontinuities. The location is also estimated to be one cell to the right, comparing with the model map.

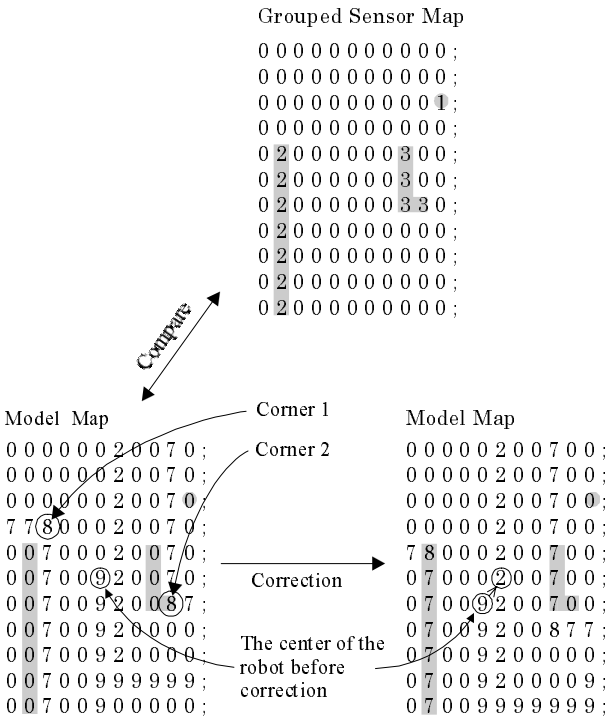


Figure 5-6: An example of map matching taken from test run #1. The shaded lines represents the occupied cells in the sensor map.

- The corner matching algorithm seeks a corner which corresponds to the corner 1 in the model map. A discontinuity in the group 2 which is one cell to the left and one cell down is estimated as a corresponding corner.
- There must be a corner at the lower left side, the corner 2. The group 3 contains an obvious corner feature and location should be shifted one cell to the right.

By gathering the information from the map matching procedure, it was determined that the location of the robot must be shifted one cell to the right and one cell up to match with the model. As the result of the position correction, the robot was actually on the model path, but the corner 2 is off by one cell upward. It is possible that the discontinuity in the group 2 is not a result of an actual discontinuity of the wall but the continuation of the wall

that was not yet in the sensor range. By averaging the errors derived from two corners, the robot position was shifted up which resulted in corner 2 no longer matching. Although the position of the robot was not adjusted correctly at this location, ANT adjusted its position as it repeated the map matching procedure at different locations.

5.4 Navigation

Six out of ten test runs were successfully operated. All the results are presented in Appendix B. The four failure cases will be discussed later in this chapter.

Figures 5-7 and 5-8 shows the result of the test run #1 as an example which had been successfully accomplished. Figure 5-7 shows the actual path superimposed on the

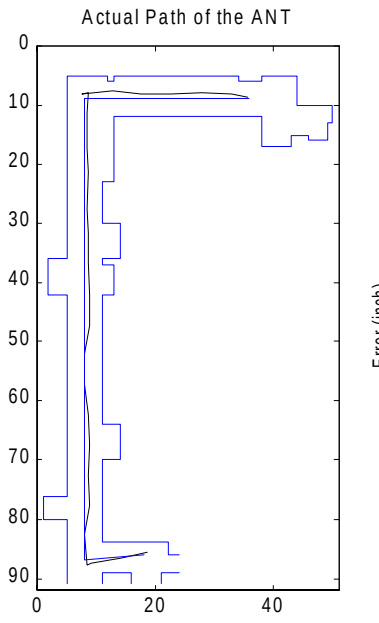


Figure 5-7: The actual path of test run #1 with the model path

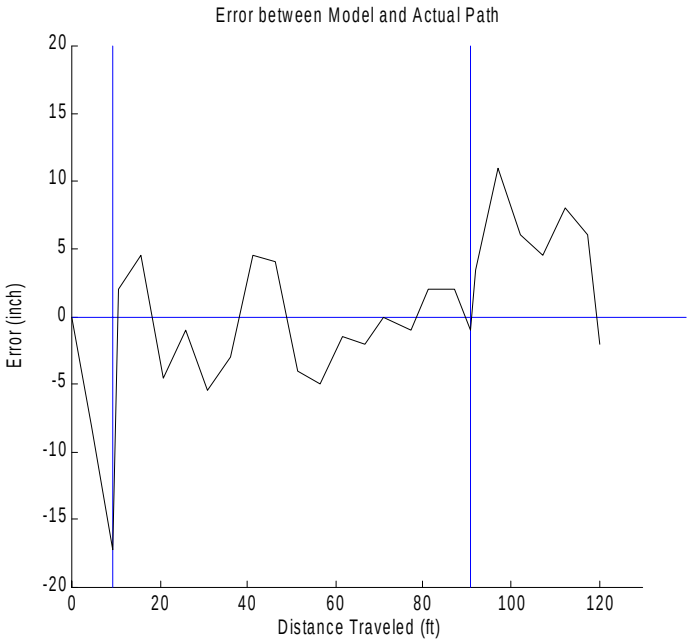


Figure 5-8: Error plot for the test run #1

desired path. ANT followed the desired path with less than 18 inches of error as shown in Figure 5-8. The largest error appeared in the first turn which is shown on the first vertical line in the error plot and it seems that ANT made a backing motion at the second turning point according to the plot although it is not capable to do it. The error is calculated as the perpendicular distance between the actual and model paths. Thus the large spikes on the turning points, which are represented as vertical lines do not simply mean the robot is making larger errors as it appears. One of the reasons for the spikes is that the model path is changed to the next straight segment at the turning points and the margin of one foot is given to the robot to accomplish the segment. The error is also considered to include human error which was caused by placing the post-it notes on the floor indicating the rear end of the vehicle not the center of the ANT's coordinates. Then the location is converted to the center of the robot afterward according to the geometry of ANT.

5.5 Failure Case Analysis

The program had to be terminated manually in four cases during the test runs. Most of the failures were caused by failure of the ultrasonic sensors to give reliable readings. The four failure test runs are categorized into three cases and discussed in this section according to the observation and log files outputted by the navigation program.

Case 1: Test Run #3 and #10

ANT bumped into a wall twice in the middle of the hallway where the model path is a long straight segment as shown in Figure 5-9. In the case of test run #3 in Figure 5-9 (left), ANT was slightly off to the right from the desired path, although it estimates that it was on the left of the desired path because of the low accuracy of the dead reckoning data and lack of ultrasonic readings. At the last recognition procedure before manual stop, ANT didn't detect any long walls on either side thus it was unable to correct the orientation.

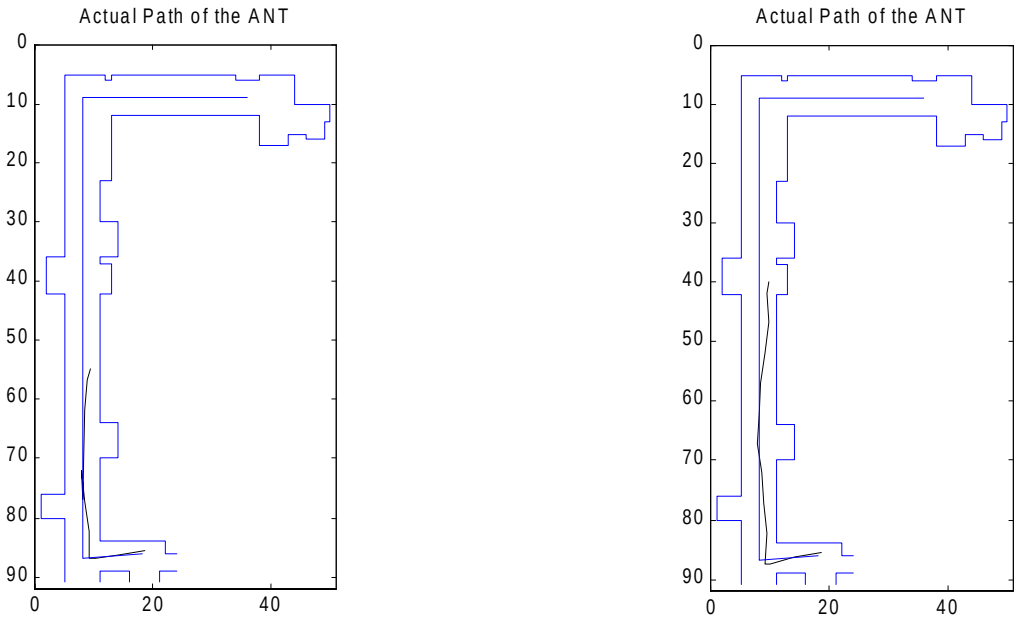


Figure 5-9: Actual path of the test run #3 (left) and #10 (right).

Therefore only the estimate angle from the encoders was taken into account and the path planning procedure made a wrong turn.

At test run #10 as shown in Figure 5-9 (right), it is more obvious why ANT lost the way. It was the most difficult place to pass, which contains two recesses, as discussed earlier in this chapter. Even though the sensors may detect both walls at the same time to build a global sensor map, ANT will not detect both walls because the distance between two walls is greater than the size of the local map which is used to perform the recognition algorithm. Thus ANT was not able to find the left side wall and the right side wall was not detected long enough to be identified as a long wall. ANT was attempting to make a left turn to correct its path to the model. But it was not enough to avoid the protruding wall section.

Case 2: Test Run #5

In this case, ANT was not able to estimate its position correctly. It estimated that the second local goal had been achieved and made a turn to the next local goal as shown in Figure 5-11. This is caused by misidentifying a corner which is the key to correct the location, in this case, y-coordinate. According to the log file, there were five occupied cells around

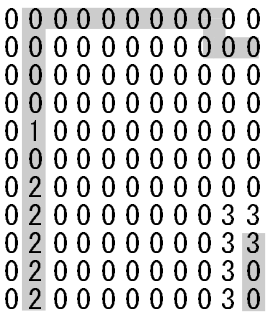


Figure 5-10: A local map taken at the second turning point of the test run #5.

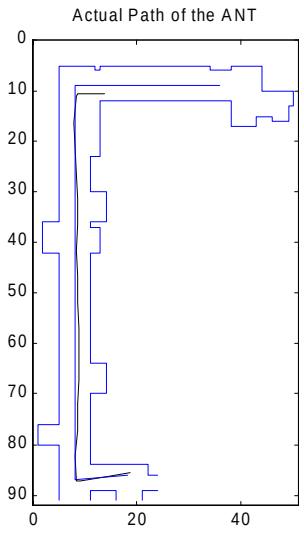


Figure 5-11: Actual path of the test run #5.

the cell which should be a corner as shown in Figure 5-10. The corner finding algorithm was not able to locate a corresponding corner from the group of the cells. Therefore the location of the robot was not corrected in this turn. Additionally, a margin of error of one foot to the local goals leads the robot to turn at a wrong location.

Case 3: Test Run #7

This test run was almost accomplished, but ANT lost the goal. The navigation procedure set a margin of 3 feet square to the goal. In other words, the navigation algorithm judged that ANT achieved a goal if it is closer than one foot to the actual goal cell. In this test run #7, ANT made a last turn just out side of the margin. Thus it made a larger turn to reach the goal. Then after a straight forward move, the map matching algorithm estimated that it passed the goal and made a almost 180 degree turn. ANT repeated turning a few times and never reached the goal. As shown in Figure 5-12, the turning radius of ANT is almost equal to the width of the hall way. Although the robot can turn on its axis, it must be in the center of the hall way to make a simple turn successfully.

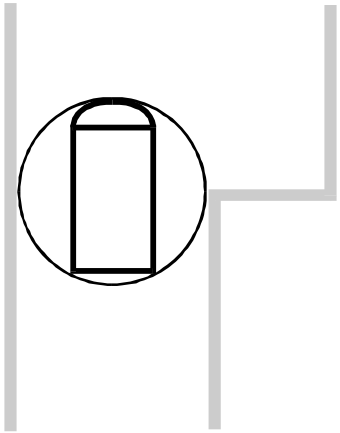


Figure 5-12: The turning radius of ANT. The gray lines are the walls in the test environment.

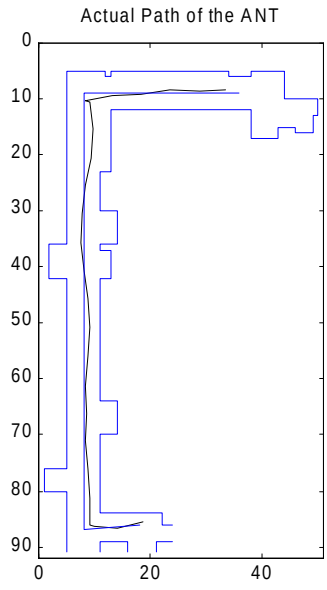


Figure 5-13: Actual path of the test run #7.

Additionally, as discussed earlier, it is preferred that the robot is parallel to the walls for better sensor mapping. When ANT repeats the almost 180 degree turns, the sensor map becomes less accurate because the certainty values may added to the wrong cells.

5.6 Recalibration of Ultrasonic Sensors

All the failure cases were caused by problems stemming from insufficient returns from the ultrasonic sensors. The ultrasonic sensor system was investigated to solve the problem and recalibrated. It was found that two channels of the multiplexer were malfunctioning and could not be repaired. Those two channels had to be disconnected from the system, and sensor No.3 and No.6 were chosen to sacrifice. Thus, the recalibration was performed on the remaining five channels. Also, a means was discovered to reset the sensor range to the distance from 6 inches to 120 inches.

First a program was built to take a hundred data points on each sensor and save them to a disk. All five sensors were programmed to fire in the same sequence as in the actual navigation program. All sensors were faced perpendicular to the wall and the distances were measured with a tape measure. Figure 5-14 shows an example of how the

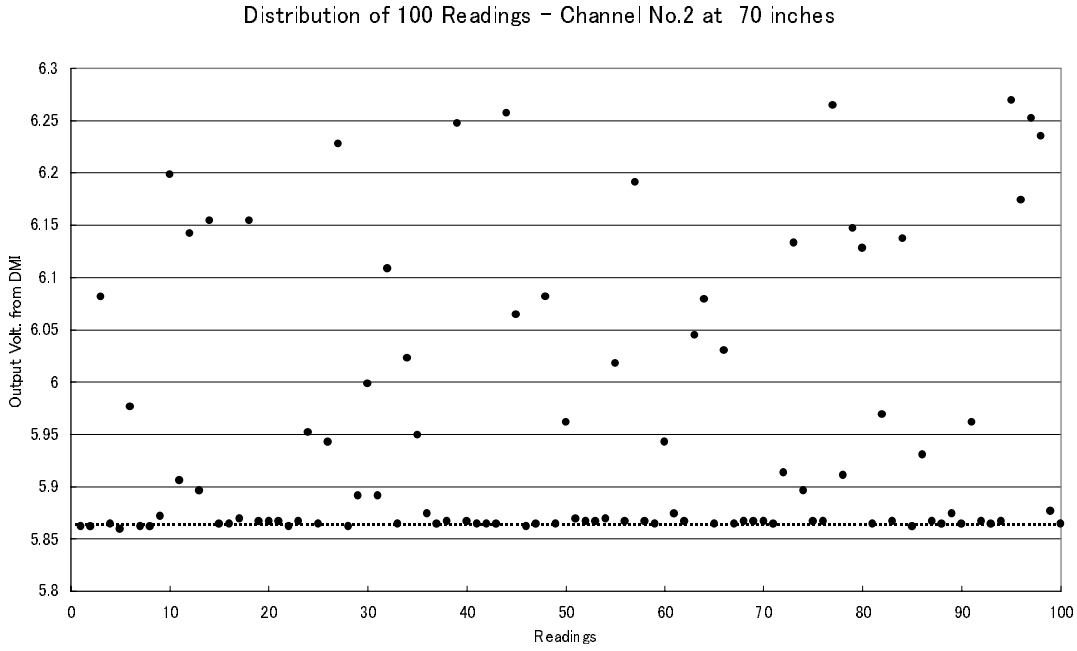


Figure 5-14: Distribution of a hundred readings while the robot is still. The dashed line represents the correct readings.

readings vary. All sensors fluctuate approximately 0.4 volts, and by observing the plots, it can be seen that noise only appears on the high side. Thus an effective approach can be to collect five readings at a time and choose the minimum value as a correct reading.

To calibrate the sensors, No.2 sensor, which was on the front, was used. One hundred readings were taken at one foot increment and recorded to a floppy disk. From the hundred readings, the output voltages were determined by fitting a line to the lowest groups of readings, as shown in Figure 5-14. The plot is shown in Figure 5-15 and the line fit is expressed as

$$\text{Distance(inch)} = 10.96 \times Volts + 6.25$$

The sensor returned false readings for the distance from 19.25 to 23.6 inches. In that range, the sensor output was exactly twice the actual reading. This problem was

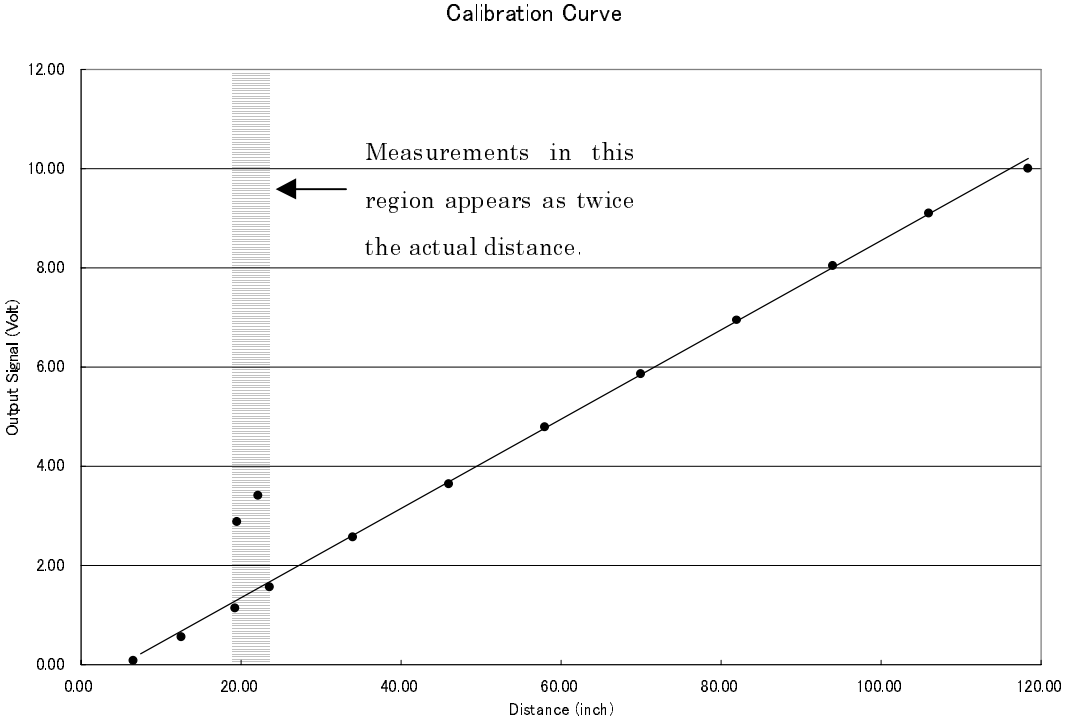


Figure 5-15: Calibration curve for the sensors. The shaded region in a distance represents the "Dead Zone."

common to all of the ultrasonic sensors, thus, it is concluded that it is a problem with the DMI hardware. Although it does not happen in other distances, this anomaly creates two spots where the distance measurement is ambiguous.

5.7 Navigation Retest

To improve the accuracy in navigation, the program was modified to include the new calibration equation and new arrangement of the ultrasonic sensors. The three front sensors had to be reduced to only one on the center due to the disconnecting two malfunctioning channels.

Additionally, new procedures were added to the navigation algorithm to avoid the manual stops. With this improved program, the robot checks the readings from each side of the sensors during the straight move. When both sensors on the same side lose their readings, it is usually because the robot is either too close to the wall or angled too much to obtain a return signal. In these cases, the robot will stop and fire its sensors to try to find the wall, or to otherwise determine its position and orientation in the corridor. If the robot can determine that it is safe to proceed, it will do so. Otherwise, it alerts the operator.

With a few test runs, it was discovered that the sparse ultrasonic data from only five sensors did not work well with Histogram In-Motion Mapping method. Many blind spots made the correlation of the sensor map to the given map too difficult to navigate the robot. Moreover, one of the PWM motor controllers failed, and further data could not be collected.

Chapter 6

Conclusion and Future Study

6.1 Conclusion from the Experimental Results

The Histogramic In-Motion Mapping technique does not seem to be the most effective method for use with sparse ultrasonic data such as in this experiment where only seven or five ultrasonic sensors were available. Some of the problems experienced were also due to the specific hardware used in this robot. The ultrasonic readings were quite noisy and unreliable partially because of the failure in the multiplexer hardware, and due to other unknown problems in the ultrasonic sensor system.

From the experimental data in Chapter 5, the navigation algorithm accomplished the task with accuracy of six out of ten test runs. With sufficient ultrasonic readings, ANT was able to build an environmental map to locate itself in the given model map and complete the navigation. The robot had to be stopped manually in four failure cases due to the incorrect position and orientation estimates.

Most of the failure cases were caused by the lack of sensor readings. The primary reason for missing ultrasonic readings was that if the robot was more than a few degrees from parallel to the wall, the ultrasonic energy reflected away from the sensor and no return was obtained. In his experiments, Borenstein used a ring of 24 ultrasonic sensors, oriented radially on a circle around his robot. With this arrangement, at least one sensor is guaranteed to be nearly perpendicular to a reflecting surface. With only five to seven sensors, there were too many cases where no data was received at all.

The problem with HIMM was compounded by the low accuracy of dead reckoning system on a tracked vehicle. A particular problem was inaccuracy in turning, especially turns more than a few degrees.

In future work,, two approaches can be applied to avoid the ultrasonic blindness. One is to use the capability of the robot to turn on its axis which is one of the advantages of the tracked vehicle. The navigation algorithm can be modified to stop and turn at places where the robot has lost a certain percent of the ultrasonic readings. The turning can be a 5 degree interval in both clock- and counterclockwise direction until the robot receives sufficient ultrasonic data. However, this modification requires more accurate data on the robot's orientation than those from the optical encoders on the motor shafts. It would also lead to a slow laborious journey, given the number of times data were lost. Another approach is to rotate the sensor itself by adding a new sensor mount which can rotate to enlarge the effective angle range of the sensors. The rotating mounts can be programmed to rotate the sensor continuously or rotate only when the mounted sensor has lost the reading.

In addition to these methods, the certainty values of each cell can be used more effectively in the map matching algorithm. Although each cell has a certainty value from zero to eight, they are divided into only two categories; the certainty values less than two for vacancy, and two or more for obstacles. Then the probability of the existence of the walls are recalculated based on the number of the grouped cells. The certainty values can be included as a variable to the formula for more accuracy.

The resolution of the maps and the size of the local maps are another subject to be

modified. The resolution was decided based on the size of the tiles on the floor, which is one foot by one foot. This resolution is fine enough to model the environment into the matrix-based map and to provide the robot as the model map. However it is too coarse when one considers the robot size and the ultrasonic sensor resolution. It can be suggested that the sensor map can be built in higher resolution. Additionally the local model map can be extracted with larger size and converted into the same resolution with the sensor map for the comparison.

6.2 Suggestions for Future Study

The most important addition to the vehicle for future work will be to replace the current ultrasonic sensing equipment with a more reliable and more comprehensive suite of sensors. As was shown in Figure 5-14, the data were noisey even under optimal conditions, i.e. the sensors were perpendicular to large stationary target. In motion, the data were even worse. Either many more sensors or means to rotate the sensors must also be added in order to gather adequate data from which to create maps.

Once the sensing system problems are overcome, there are many functions that can be added to the ANT system. One important area of research is an obstacle avoidance algorithm. It was assumed here that there were no unknown obstacles in the operating environment because the hallway is too narrow for ANT to take an avoidance trajectory. For operation in larger fields, an obstacle avoidance procedure becomes an important issue to include in the robot navigation. Even in the narrow hallway that is used in this project, an obstacle detection program can be included to analyze if the unknown obstacles affect

the robot trajectory. If an obstacle is blocking the trajectory, the robot can simply stop and warn an operator.

It would also be useful to expand the number of features the ANT could recognize. With more sensors and more complete/complex maps, one could include such features as curved walls, doorways, small objects such as chairs or furniture, and other items in the maps. The resolution of the existing sensor system was too coarse to make these refinements possible in this work.

References

1. Seneviratne, L. D., et al. "Triangulation-Based Path Planning for a Mobile Robot." Journal of Mechanical Engineering Science 211C (1997): 365-71.

2. McIntosh, T.A. "A Real-Time Architecture for Obstacle Avoidance and Path Planning for West Virginia University's Mobile Robot Platform ANT." Thesis. West Virginia University, 1994.

3. Kitano, M., and Kuma, M. "An analysis of horizontal plane motion of tracked vehicles." Journal of Terramechanics 14 (1977): 211-25.

4. Kitano, M., and Jyozaki, H. "A Theoretical Analysis of Steerability of Tracked Vehicles." Journal of Terramechanics 13 (1976): 133-41.

5. Wong, J. Y., and Preston-Thomas, J. "Parametric Analysis of Tracked Vehicles Using an Advanced Computer Simulation Model." Proc Instn Mech Engrs 200 (1986): 101-14.

6. Acker, F. E., and Krishnan, R. "A Model of Turning and Tramming Motion of a Continuous Miner," 11th International Mining Electrotechnology Conference (1992): 133-41.

7. Borenstein, Johann, and Koren, Yoram. "Error Eliminating Rapid Ultrasonic Firing for Mobile Robot Obstacle Avoidance." IEEE Transactions on Robotics and Automation 11 (1995): 132-38.

8. Min, Byung-Kwon, et al. "Exploration of a Mobile Robot Based on Sonar Probability Mapping." Journal of Dynamic Systems 118 (1996): 150-57.

9. Borenstein, J., et al. "Mobile Robot Positioning: Sensors and Techniques." Journal of Robotic Systems 14 (1997): 231-49.

10. Yagi, Yasushi, et al. "Map-Based Navigation for a Mobile Robot with Odmnidirectional Image Sensor COPIS." IEEE Transactions on Robotics and Automation 11 (1995): 634-48.

11. Borenstein, Johann, and Koren, Yoram. "Histogramic In-Motion Mapping for Mobile Robot Obstacle Avoidance." IEEE Transactions on Robotics and Automation 7 (1991): 535-39.

12. Borenstein, Johann, and Koren, Yoram. "The Vector Field Histogram – Fast Obstacle Avoidance for Mobile Robot." IEEE Transactions on Robotics and Automation 7 (1991): 278-88.

13. Gonzalez, E., et al. "Uncertainty Treatment in a Surface Filling Mobile Robot." Lecture Notes in Computer Science 1093 (1996): 294-306.

14. Hebert, P., et al. "Probability Map Learning: Necessity and Difficulties." Lecture Notes in Computer Science 1093 (1996): 307-20.

15. Leonard, J. J., et al. "Dynamic Map Building for an Autonomous Mobile Robot." The International Journal of Robotics Research 11 (1992): 286-97.

16. Durrant-Whyte, H. F. "Where am I?; A Tutorial on Mobile Vehicle Localization." The Industrial Robot 21 (1994): 11-6.

17. Oriolo, Giuseppe, et al. "Real-Time Map Building and Navigation for Autonomous Robots in Unknown Environments." IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics 28 (1998): 316-32.

18. Ko, Joong Hyup, et al. "A Method of Acoustic Landmark Extraction for Mobile Robot Navigation." IEEE Transactions on Robotics and Automation 12 (1996): 478-85.

19. Fujimori, Atushi, et al. “Adaptive navigation of mobile robots with obstacle avoidance.” IEEE Transactions on Robotics and Automation 13 (1997): 596-602.

20. Tenney, Adrian M. “Optimal Course Correction Control for a Dual Tracked Mobile robot.” Thesis. West Virginia University, 1997.

21. Banta, Larry E., et al. "A Reduced-Order, Extended Kalman Filter for AGV Navigation." Proceedings of the ASME Winter Annual Meeting (1987): 13-18.

22. Boley, Daniel L., et al. “Recursive Total Least Squares: An Alternative to Using the Discrete Kalman Filer in Robot Navigation.” Lecture Notes in Computer Science 1093 (1996): 221-33.

Appendix A

Equipment List

A.1 List of the Equipment used in the robot

- ADA2110 – Real Time Devices, Inc.
  P.O. Box 906
  State College, Pennsylvania 16804
  Phone: (814) 234-8087
  FAX: (814) 234-5218

  12-bit, 20 microsecond A/D Converter
  ±5, ±10, or 0 to +10 volt Analog Input Range
  Three 16-bit Timer/Counters, on-board 8 MHz Clock
  2 Fast Settling, Double-Buffered, Analog Output Channels; ±5, ±10, 0 to +5, or 0 to +10 volt output range
  8 Differential/16 Single-Ended Analog Input Channels

- DT2817 – Data Translation, Inc.
  100 Locke Drive
  Marlborough, MA 01752-1192

  Digital-to-Digital Interface board
  32 Digital I/O Channels

- HCTL-2016 – Hewlett Packard
  Phone: 1-800-235-0312

  Quadrature Decoder/Counter Interface IC
  14 MHz Clock Operation
  Full 4X Decode
  12 or 16 bit binary Up/Down Decoder

- DMI – Contaq Technologies Corporation
  15 Main Street
  Bristol, Vermont 05443
  Phone: (802) 453-3332
  FAX: (802) 453-4250

  Distance Measurement Instrument
  Manual or Computer Programmable
  Analog Outputs and Relays
  Serial ASCII, 9600 Baud I/O
  4 Digit LED Display

- UDM-MUXP – Contaq Technologies Corporation
  15 Main Street
  Bristol, Vermont 05443
  Phone: (802) 453-3332
  FAX: (802) 453-4250

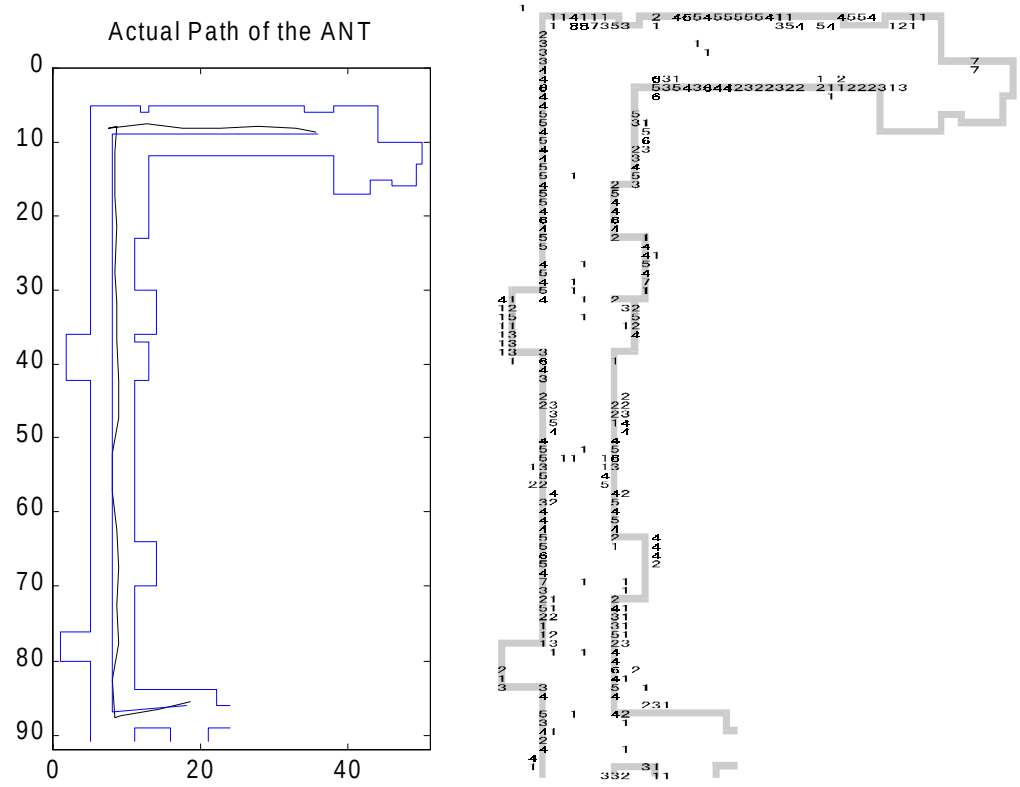
  Standalone Multiplexer
  Expands systems from 1 to 14 transducers
  Standalone Package with Independent Power Supply
  Program or Manual Transducer Channel Selection

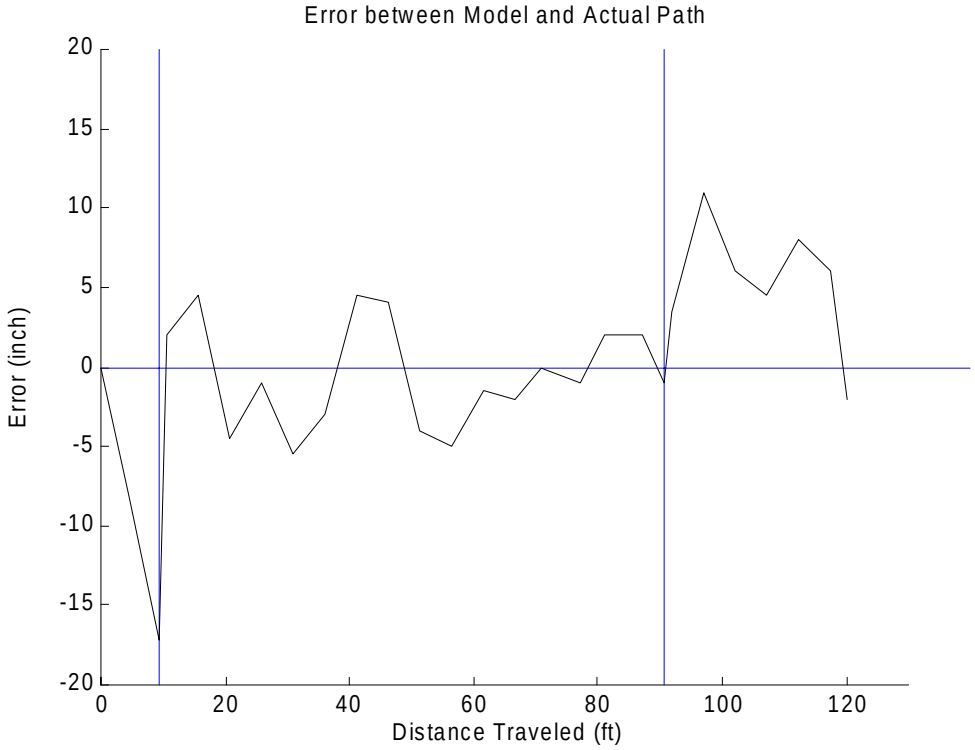
Appendix B

Results of Ten Test Runs

Test Run #1

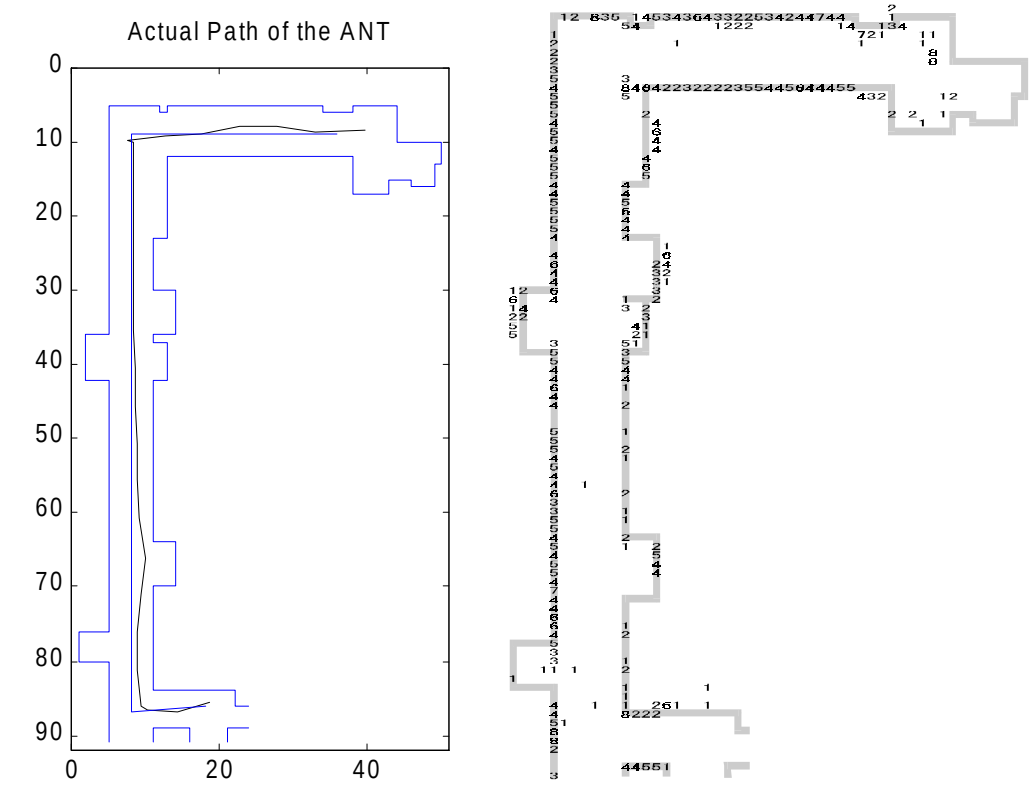
Sensor map superimposed on the actual map

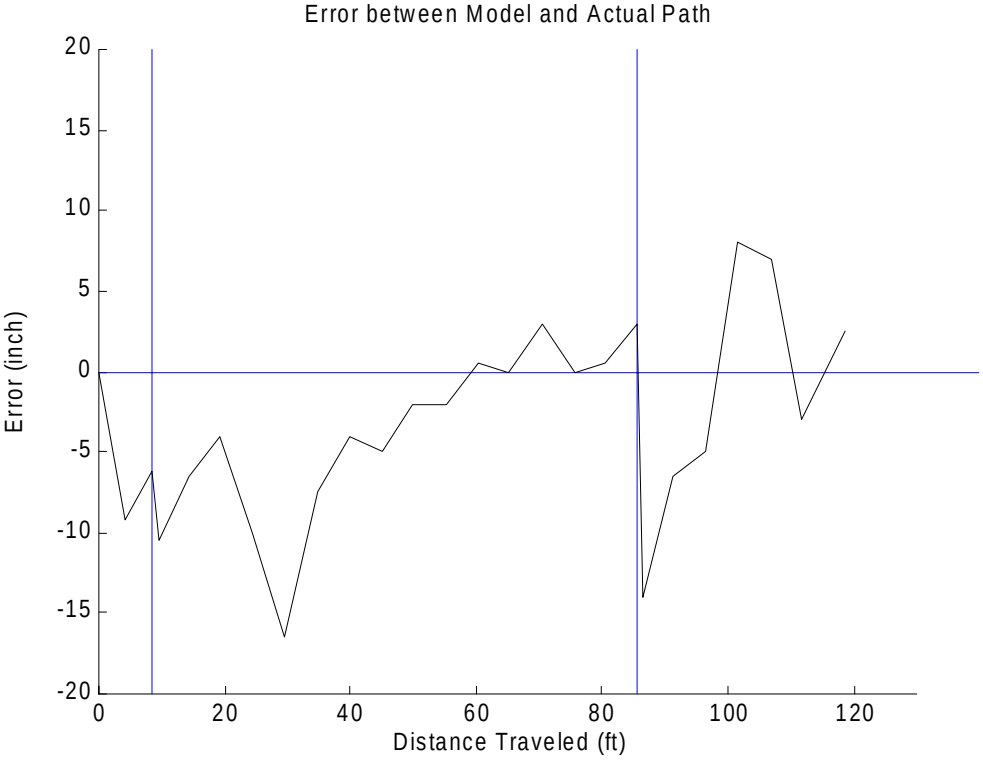




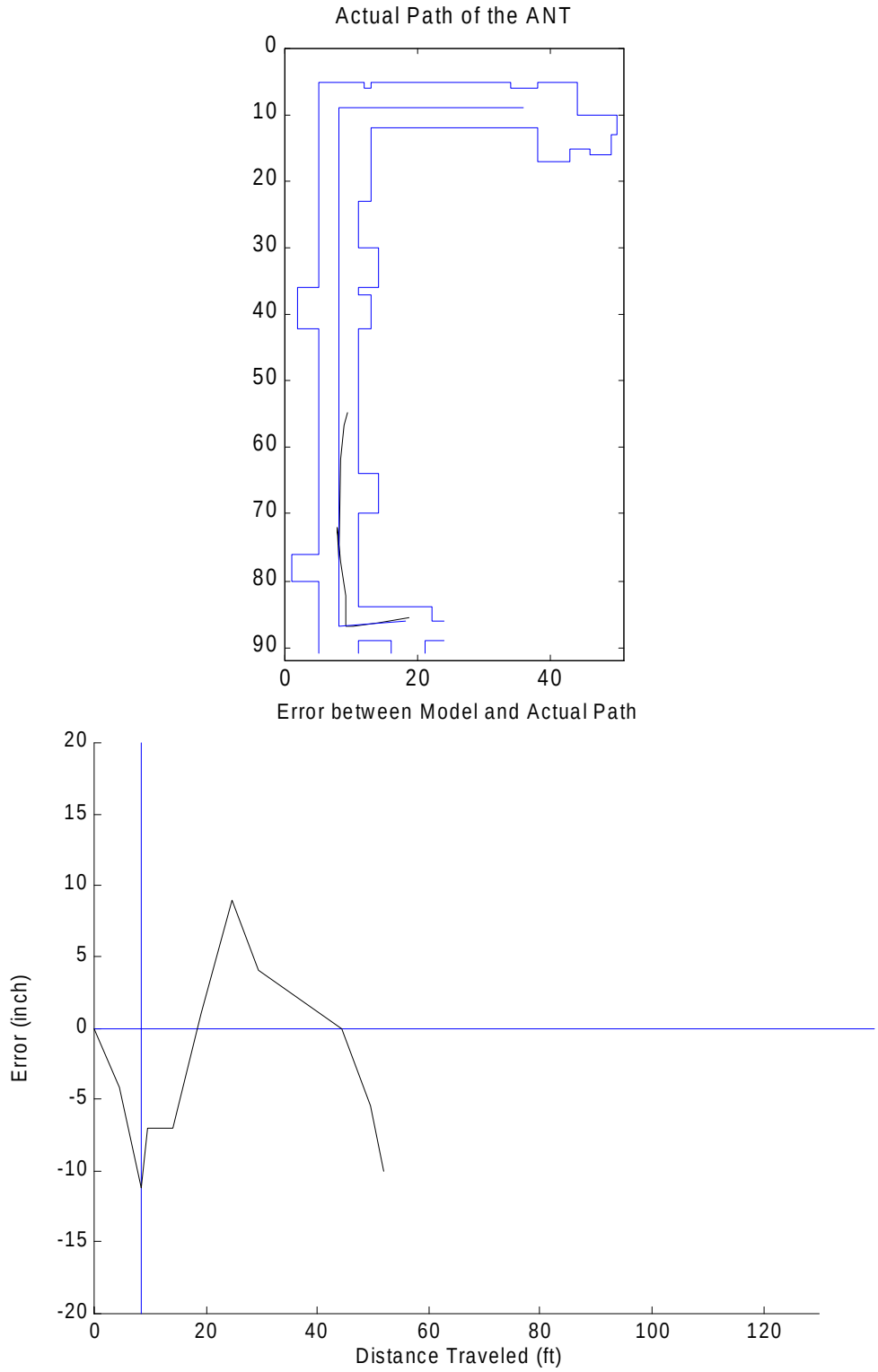
Test Run #2

Sensor map superimposed on the actual map



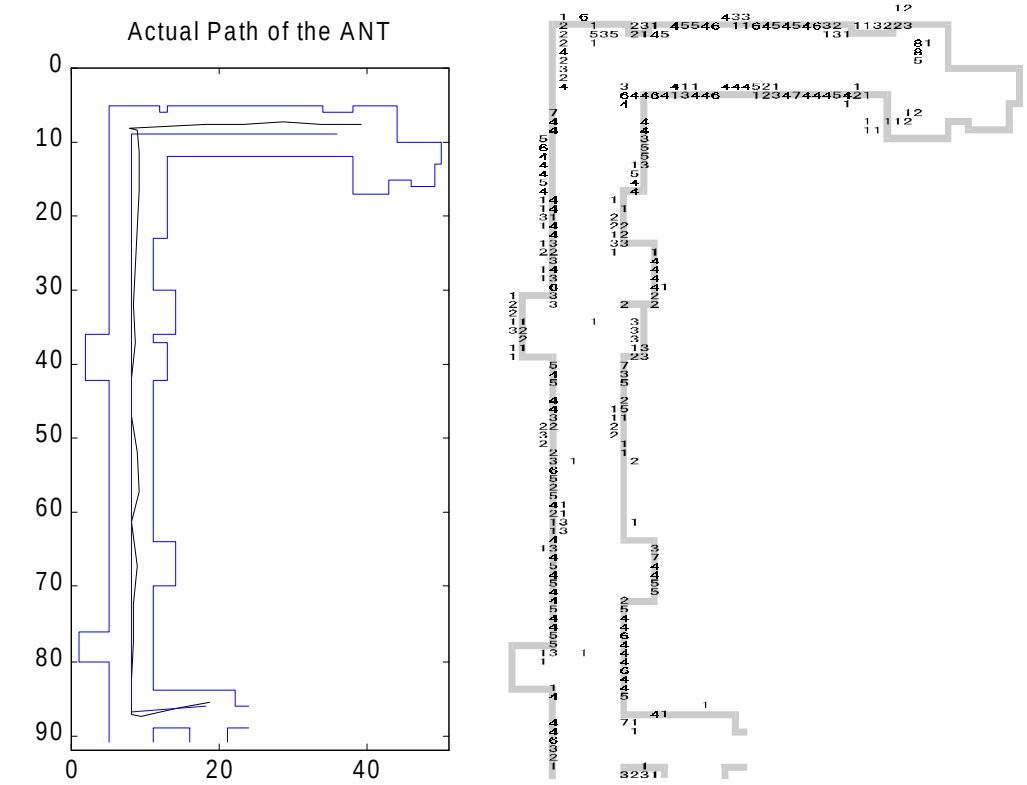


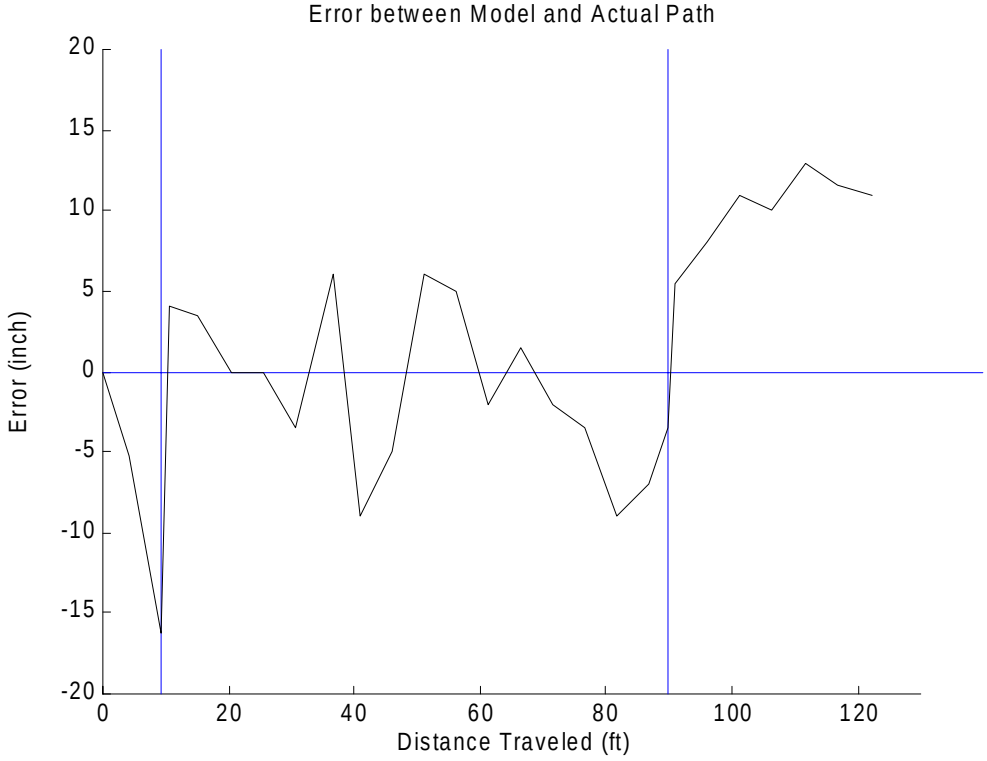
Test Run #3


Actual Path of the ANT
0
10
20
30
40
50
60
70
80
90
0
20
40
Error between Model and Actual Path
20
15
10
5
0
-5
-10
-15
-20
Error (inch)
0
20
40
60
80
100
120
Distance Traveled (ft)

Test Run #4

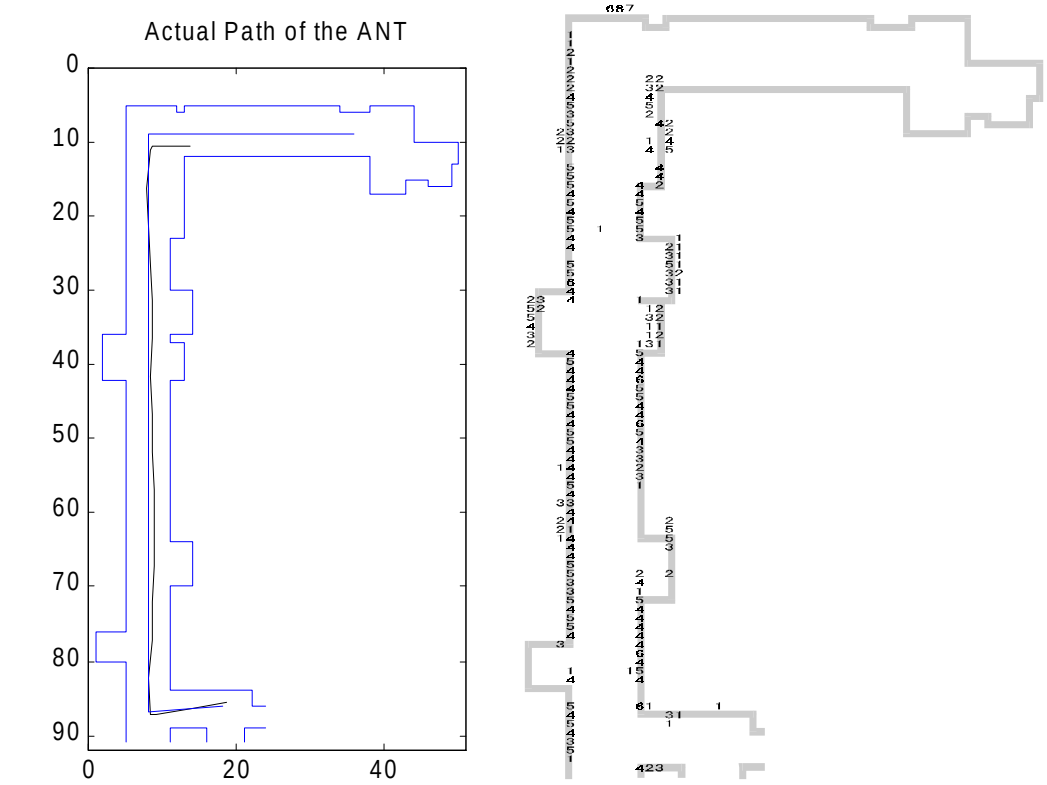
Sensor map superimposed on the actual map

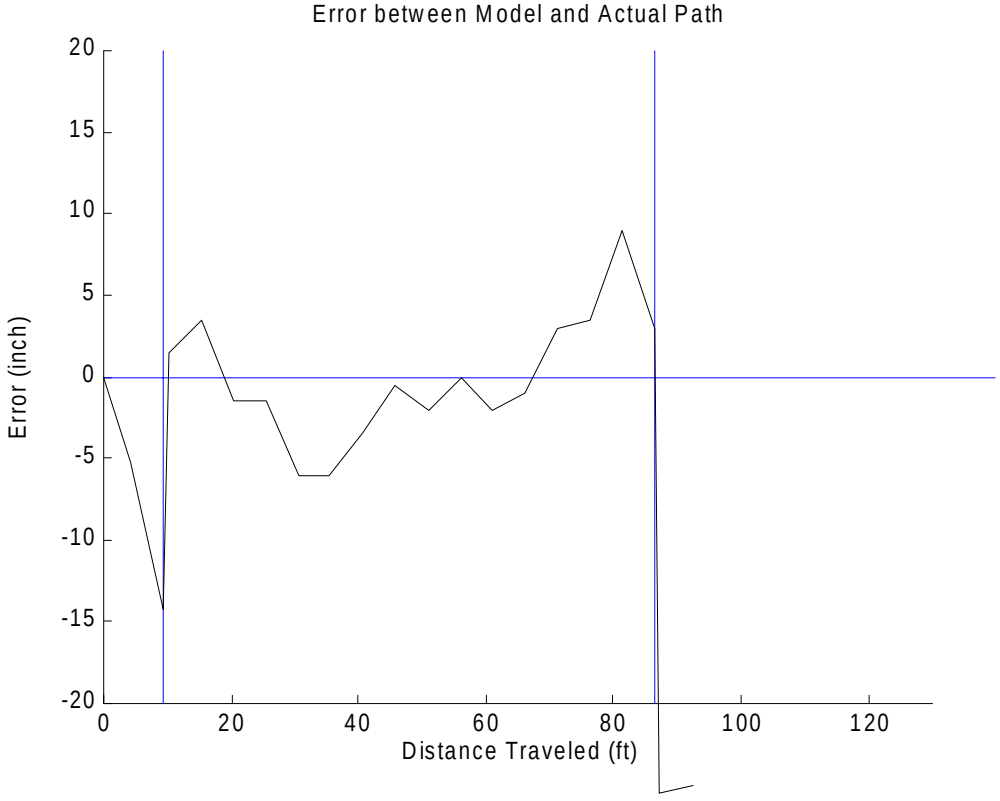




Test Run #5

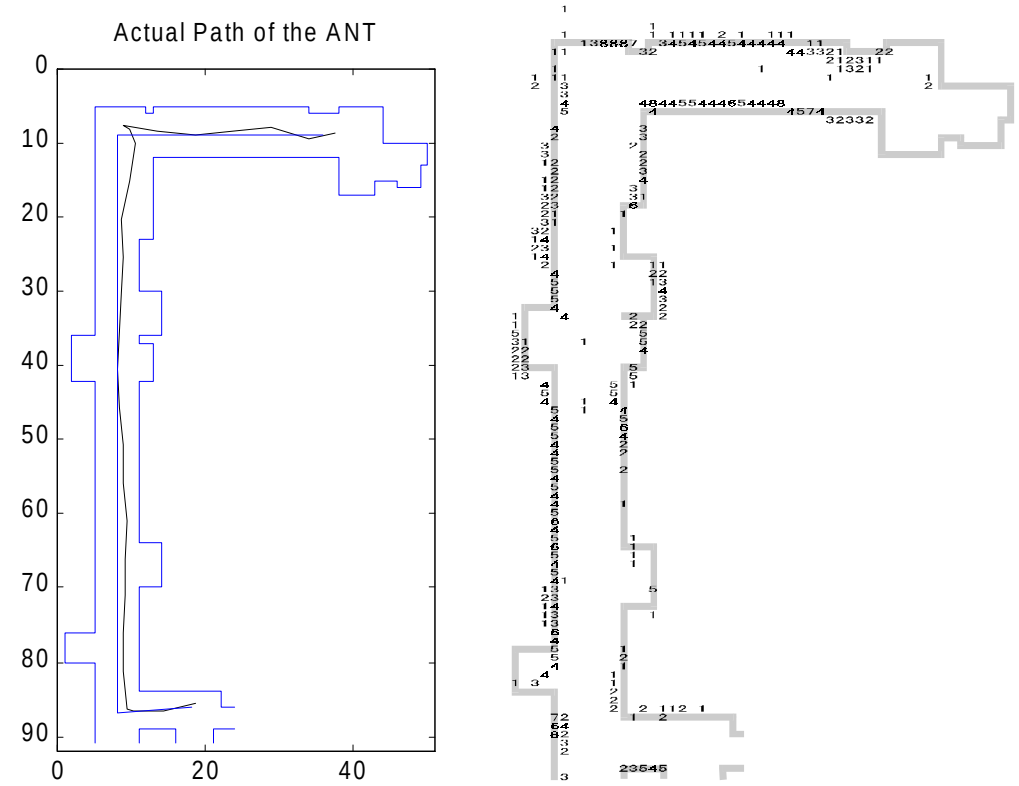
Sensor map superimposed on the actual map

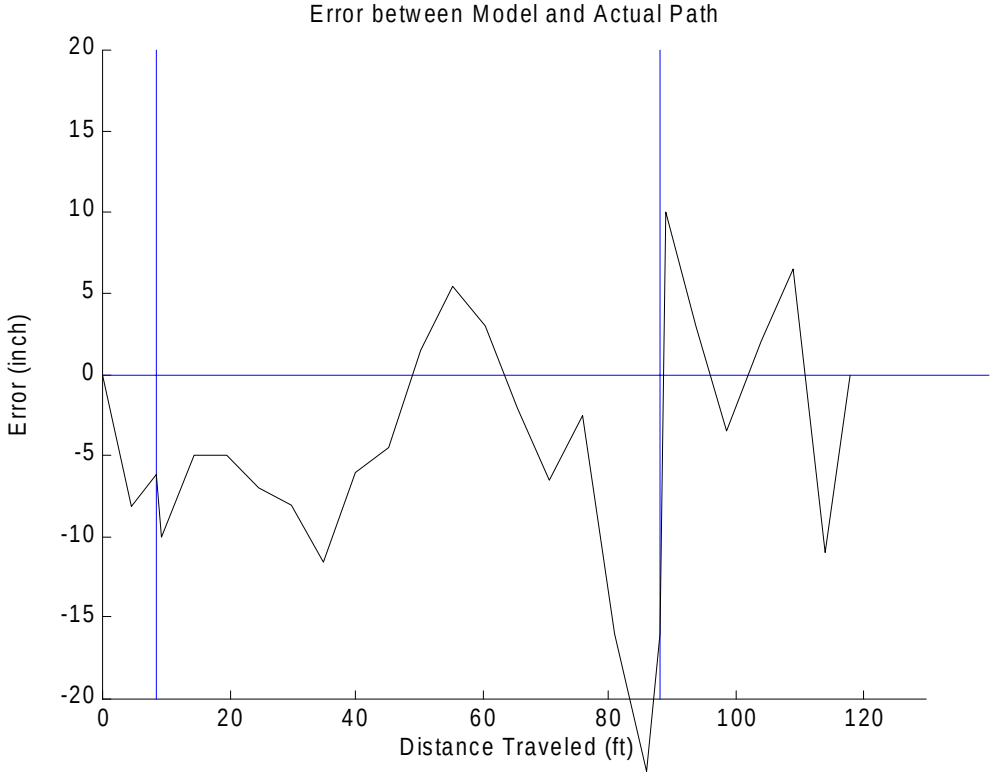




Test Run #6

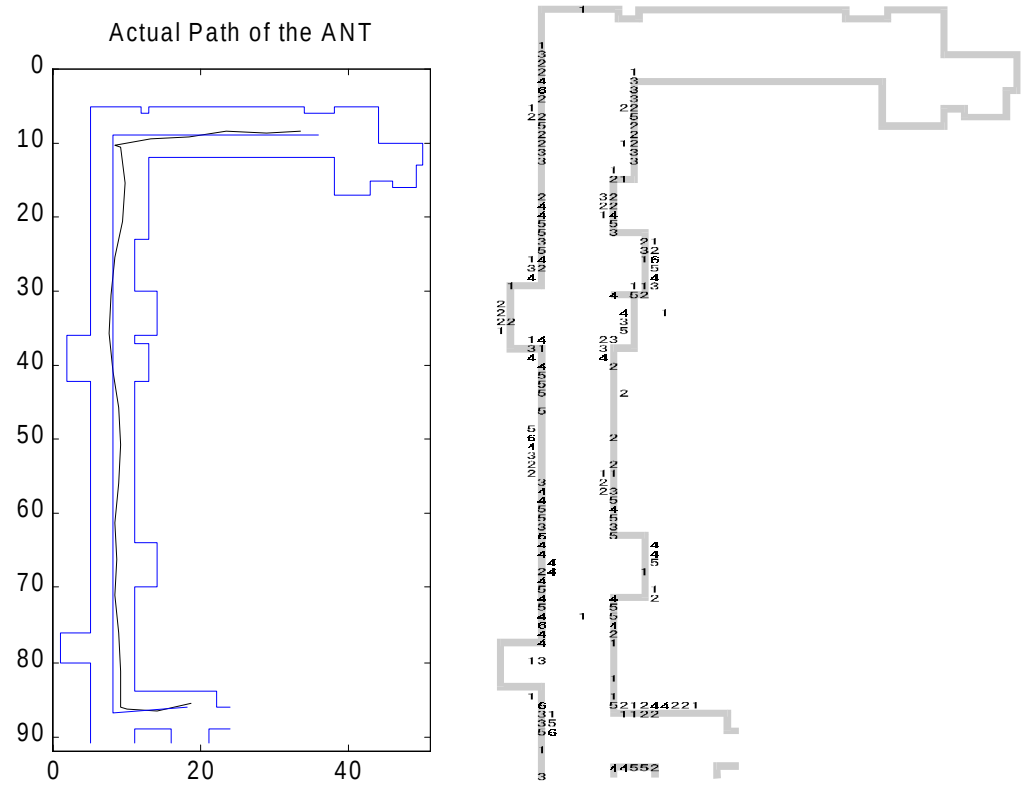
Sensor map superimposed on the actual map

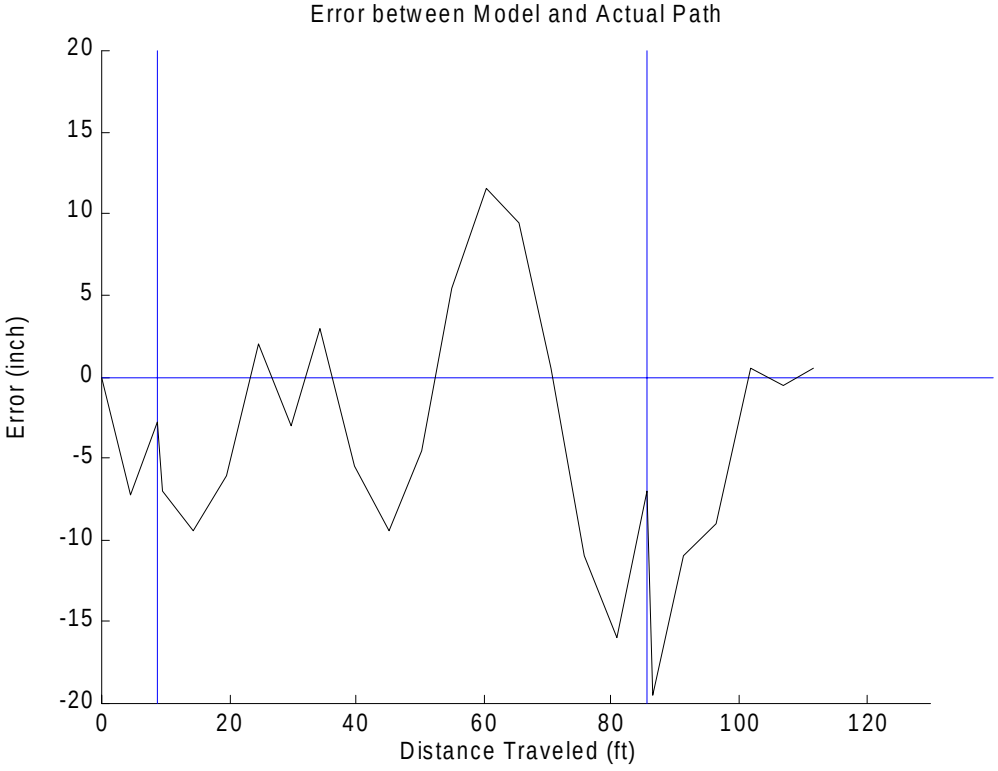




Test Run #7

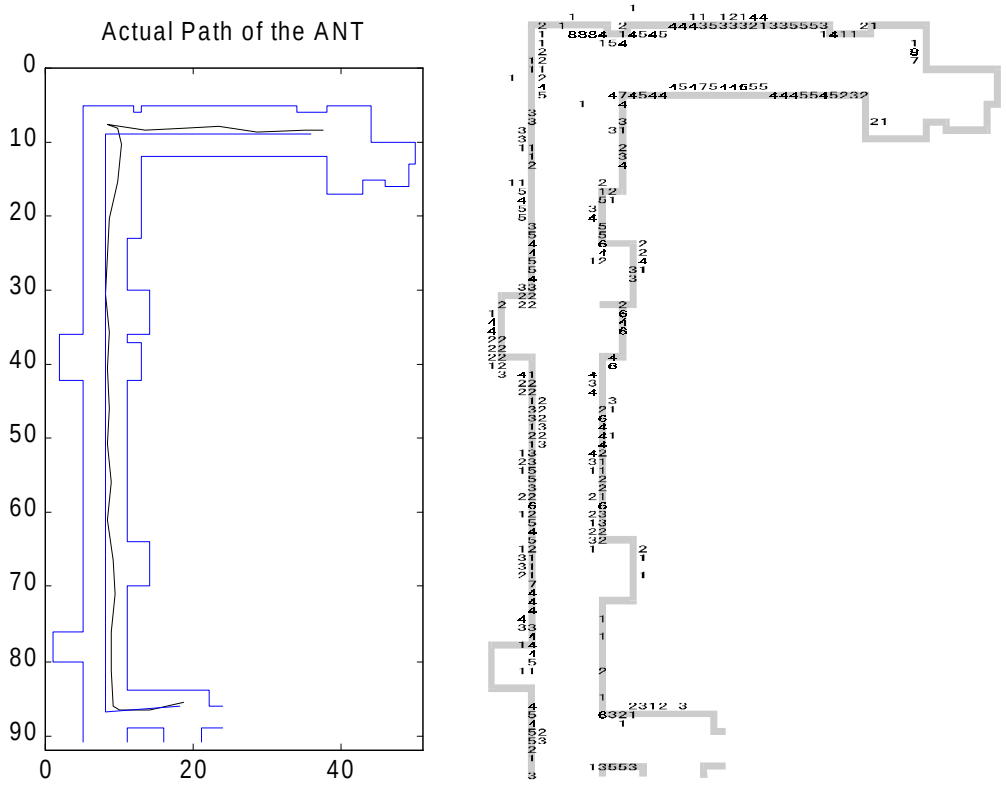
Sensor map superimposed on the actual map

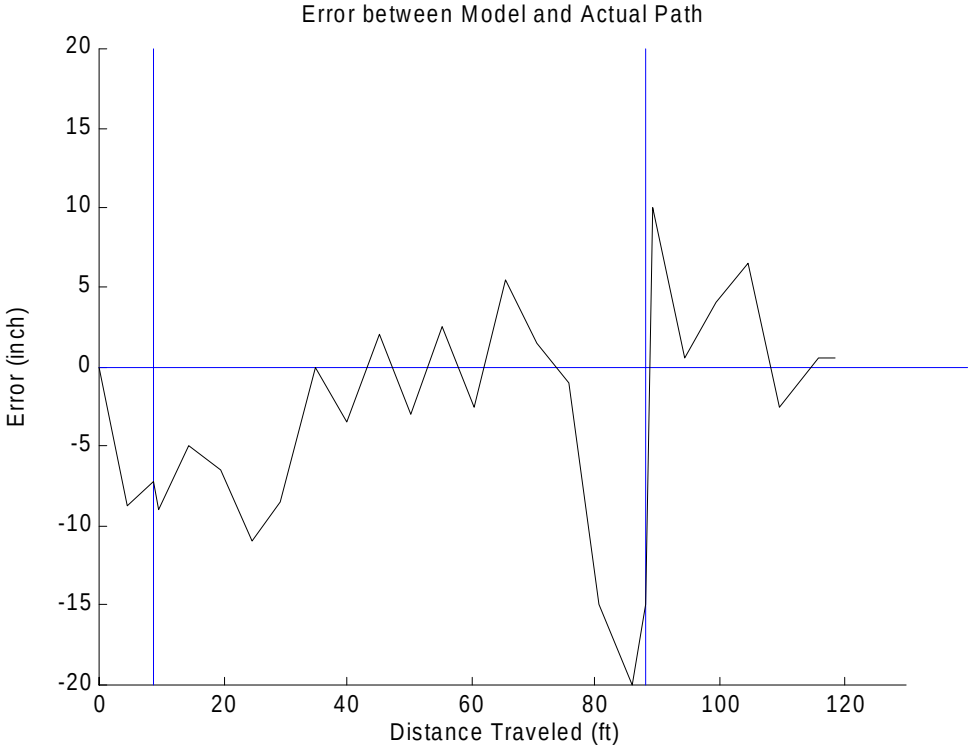




Test Run #8

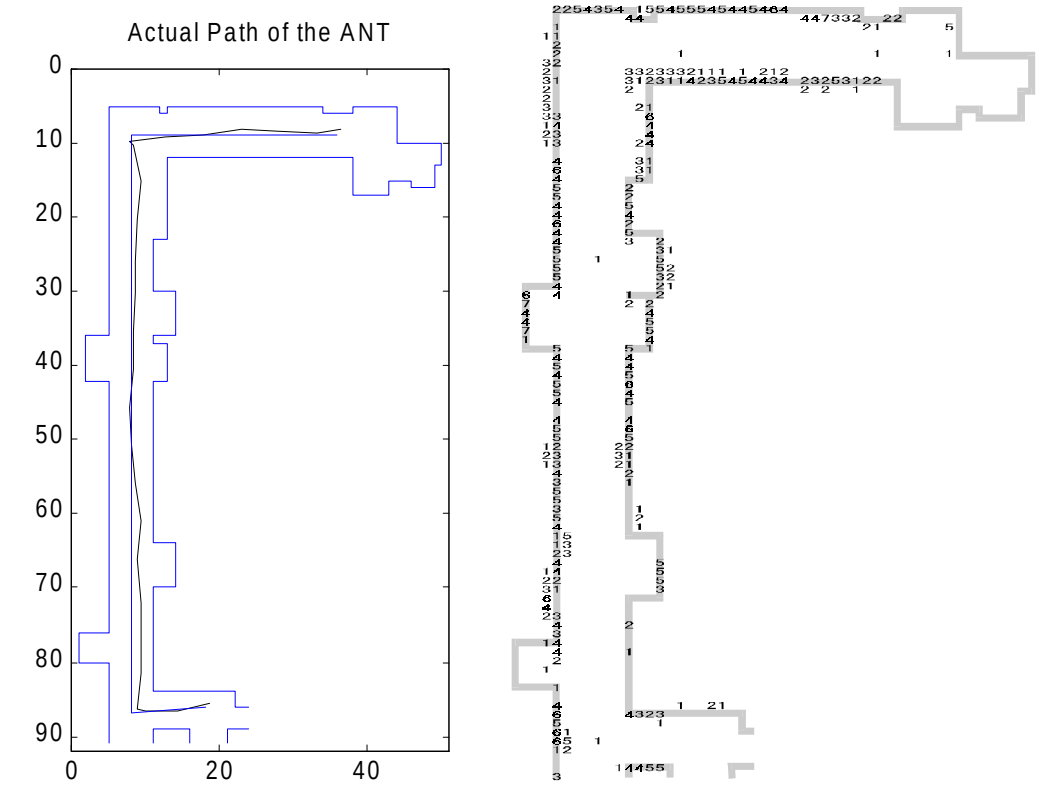
Sensor map superimposed on the actual map

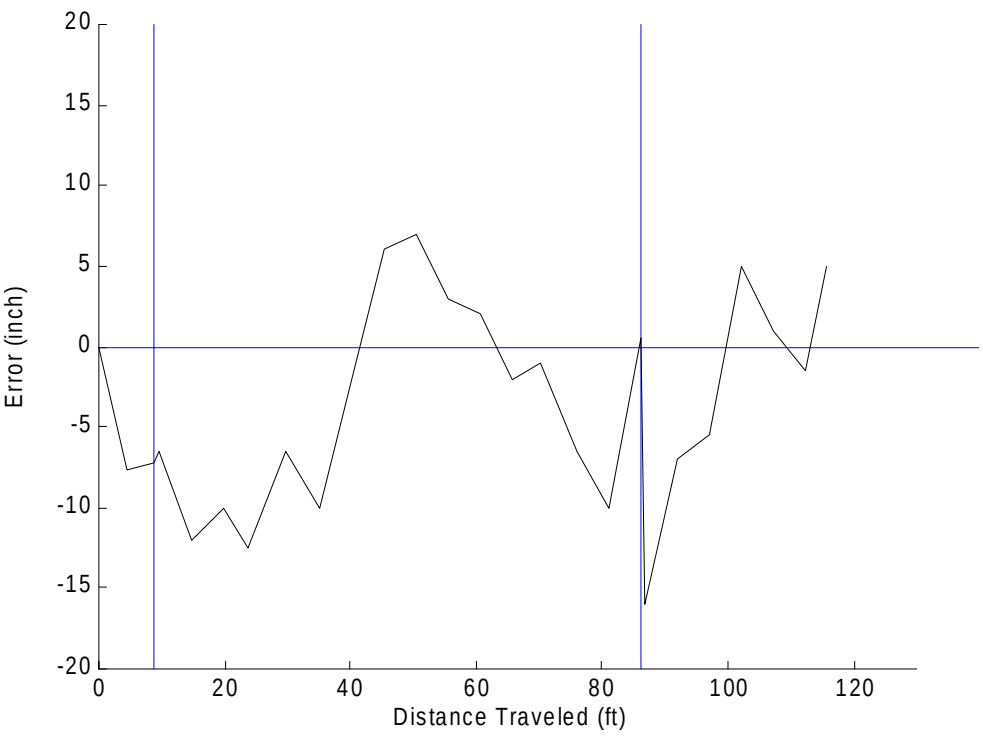




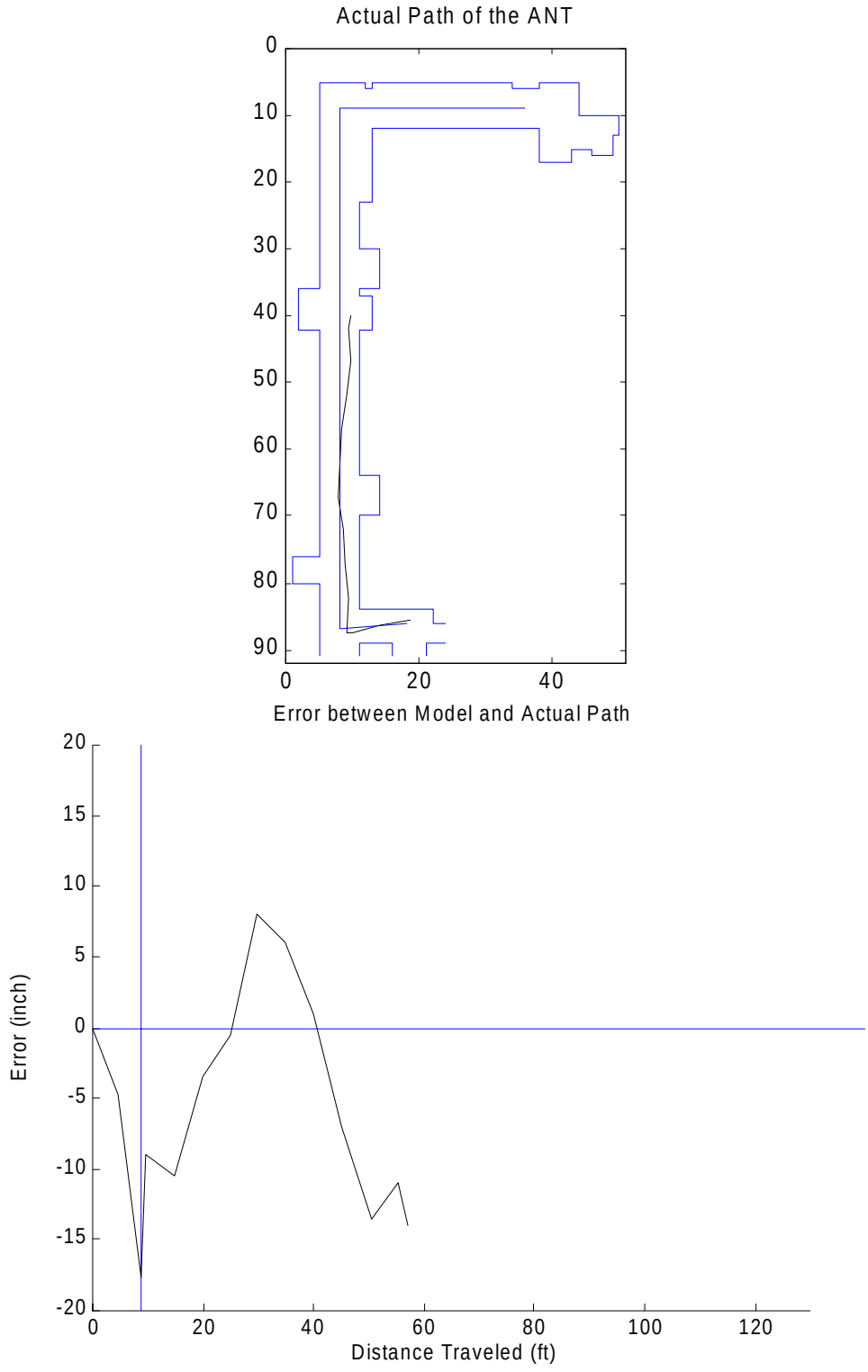
Test Run #9

Sensor map superimposed on the actual map





Test Run #10


Actual Path of the ANT
0
10
20
30
40
50
60
70
80
90
0
20
40
Error between Model and Actual Path
20
15
10
5
0
-5
-10
-15
-20
Error (inch)
0
20
40
60
80
100
120
Distance Traveled (ft)

Appendix C

Source Codes

C.1 Navigation Program Source Code

```
/*
    File Name:        navi.c++
    Programmed by: Akihiko Baba
    Last Build:       8/31/98
*/

#include<windows.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include"ADA2110.h"

// Definitions for DT2817
#define DTBA                0x228 //Base Address of DT2817
#define CHANNEL_A           DTBA + 1
#define CHANNEL_B           DTBA + 3
#define CONTROL             DTBA + 2
#define CTRL_WORD           DTBA

// Control Words for DT2817 to control two HCTL chips
#define Board_Initialize    2
#define Latch_High          0
#define Latch_Low           18
#define Done                9

// File Definitions
#define IN_PATH             "A:¥¥path.map"
#define IN_GIVEN            "A:¥¥givenmap.map"
#define OUT_SENSOR          "D:¥¥builtmap.map"
#define OUT_PATH            "D:¥¥pathmap.map"
#define OUT_DATA            "D:¥¥logdata.txt"

// Other Definitions
#define GOAL                3   // 3 at the goal   //
#define STEP                60  // Interval between Map Matching
#define RADIUS              16  // Turning radius of ANT

FILE *outpointer_data;

int     ChA, ChB;               // Encoder readings
int     ChA_0, ChB_0;           // Initial values of encoder readings
int     deltaChA, deltaChB;     // Differences in encoder readings
int     iX, iY;                 // Integer value of coordinates
int     SensorNo=0;             // Sensor index number
```

```
float   vel, angle;                 // Velocity and Orientation of ANT
foat    xMAP, yMAP, tMAP;           // Coordinates and Orientation of ANT in Global Coordinate System
float   DistTravel, AngleTravel;    // Total Distance and Orientation of ANT
float   SensorRead[7];              // Sensor Readings
int     LMAP[51][92]={0}, sLMAP[11][11]={0};    // Global and Local Sensor Map
int     GMAP[51][92], sGMAP[11][11]={0};        // Global and Local Model Map
int     PATH[2][GOAL+1];                        // Turning Points Coordinates


/*
    round

    This is a procedure which performs a standard rounding.
    It takes a double variable and returns a rounded int variable.
*/

int round(double num)
{
    int inum;                       // Integer number
    double FLOOR, CEIL;             // floor and ceil number of given num
    FLOOR=num-floor(num);
    CEIL=ceil(num)-num;
    if (FLOOR>CEIL|CEIL==0)
        inum=int(ceil(num));
    else if (FLOOR<CEIL|FLOOR==0)
        inum=int(floor(num));
    else if (FLOOR==CEIL)
    {
        if (num>=0)
            inum=int(ceil(num));
        else if (num<0)
            inum=int(floor(num));
    }
    return inum;
}


/*
    EncoderInitialize

    This procedure initializes the DT2817 board for Encoder
    reading and provides initial values for each variables.
*/

void EncoderInitialize()
{
```

```
    int     Ahigh, Bhigh;       // MSB of the encoder readings
    int     Alow, Blow;         // LSB of the encoder readings
    float   radius;             // Turning radius

    _outp(DTBA, Board_Initialize);

    _outp(CONTROL, Latch_High);
    TimeDelay(10);
    Ahigh = _inp(CHANNEL_A);
    Bhigh = _inp(CHANNEL_B);
    _outp(CONTROL, Latch_Low);
    TimeDelay(10);
    Alow = _inp(CHANNEL_A);
    Blow = _inp(CHANNEL_B);
    _outp(CONTROL, Done);

    ChA_0 = Ahigh*256+Alow;
    ChB_0 = Bhigh*256+Blow;

    deltaChA = 0;
    deltaChB = 0;
    vel = 0;
    radius = float(7.86*exp(4.5346));
    angle = 0;
}


/*
    Encoders

    This subroutine reads the encoders on both tracks and
    computes distance traveled and turning radius from the
    difference between two readings.
*/

void Encoders()
{
    int     Ahigh, Bhigh;       // MSB of the encoder readings
    int     Alow, Blow;         // LSB of the encoder readings
    int     ChA_old, ChB_old;   // Last readings
    int     sign;               // either +1 or -1
    float   radius, AngVel;     // Turning radius and angular velocity of ANT

    ChA_old = ChA;
    ChB_old = ChB;
    _outp(CONTROL, Latch_High);
```

```
TimeDelay(10);
Ahigh = _inp(CHANNEL_A);
Bhigh = _inp(CHANNEL_B);
_outp(CONTROL, Latch_Low);
TimeDelay(10);
Alow = _inp(CHANNEL_A);
Blow = _inp(CHANNEL_B);
_outp(CONTROL, Done);
ChA = (Ahigh)*256+(Alow) - ChA_0;
ChB = (Bhigh)*256+(Blow) - ChB_0;

deltaChA = ChA - ChA_old;
deltaChB = ChB - ChB_old;
vel = float(deltaChA + deltaChB)*6/90;   // in inch

if (vel <-50 | vel >50)    // ----------------   In case of RollOver
    vel = 0;               // ----------------   Ignore

DistTravel = DistTravel+vel;   // in inch

if (deltaChA>0&deltaChB>0)
    angle = 0;
else if (deltaChA<0&deltaChB>0)
{
    sign = -1;
    AngVel = float(deltaChB-deltaChA)*6/90;
    radius = RADIUS;

    angle = AngVel/radius*float(sign);

    if (angle <-10 | angle >10)  // ----------------   In case of RollOver
        angle = 0;               // ----------------   Ignore
}
else if (deltaChA>0&deltaChB<0)
{
    sign = 1;
    AngVel = float(deltaChA-deltaChB)*6/90;
    radius = RADIUS;

    angle = AngVel/radius*float(sign);

    if (angle <-10 | angle >10)  // ----------------   In case of RollOver
        angle = 0;               // ----------------   Ignore
}
else
    angle=0;
```

```
}


/*
    LoadGlobalMap

    This procedure reads two text files from the floppy drive.
    One is the global map of the environment and the other contains
    the initial position and orientation and the coordinates which
    the robot must turn.
*/

void LoadGlobalMap()
{
    FILE *inpointer_map, *inpointer_path;
    int mx, my;                 // Coordinates of Map
    inpointer_map = fopen(IN_GIVEN,"r");
    for(my=0;my<92;my++)
    {
        for(mx=0;mx<51;mx++)
        {
            fscanf(inpointer_map,"%i",&GMAP[mx][my]);
        }
    }
    fclose(inpointer_map);

    inpointer_path = fopen(IN_PATH,"r");
    fscanf(inpointer_path,"%f",&tMAP);
    for(my=0;my<=GOAL;my++)
    {
        for(mx=0;mx<2;mx++)
        {
            fscanf(inpointer_path,"%i",&PATH[mx][my]);
        }
    }
    xMAP=float(PATH[0][0]);
    yMAP=float(PATH[1][0]);
    iX = round(xMAP);
    iY = round(yMAP);
    fclose(inpointer_path);
}


/*
    SaveBuiltMap
```

```
    This procedure saves the sensor map built from the sensor readings
    and the path map which contains the estimated path of the robot.
*/

void SaveBuiltMap()
{
    FILE *outpointer_path, *outpointer_sensor;
    int mx, my;                         // Coordinates of Map
    outpointer_sensor = fopen(OUT_SENSOR,"w");
    outpointer_path = fopen(OUT_PATH,"w");

    for(my=0;my<92;my++)
    {
        for(mx=0;mx<51;mx++)
        {
            fprintf(outpointer_sensor,"%i ",LMAP[mx][my]);
            fprintf(outpointer_path,"%i ",GMAP[mx][my]);
        }
        fprintf(outpointer_sensor,"¥n");
        fprintf(outpointer_path,"¥n");
    }
    fclose(outpointer_sensor);
    fclose(outpointer_path);
}


/*
    MapBuild

    This procedure builds the environment map according to the
    sensor reading.
*/

void MapBuild()
{
    float phi;                      // Half of the ANT's orientation
    phi = angle/2;
    tMAP = tMAP + 2*phi;
    xMAP = xMAP - ((vel/12)*float(sin(tMAP)));
    yMAP = yMAP - ((vel/12)*float(cos(tMAP)));

    iX = round(xMAP);
    iY = round(yMAP);
    GMAP[iX][iY]=9;
}
```

```
/*
    SmallMaps

    This procedure builds a two 11 by 11 map matrices which
    surrounds the robot at the given position, one is from
    the given global map and other is from the sensor map.
*/

void SmallMaps()
{
    int LocalX=0, LocalY=0, x, y;              // Coordinates indices
    LocalX=0;

    LocalY=0;
    for (y=iY-5;y<=iY+5;y++)
    {
        LocalX=0;
        for (x=iX-5;x<=iX+5;x++)
        {
            if (x<0 | x>=51)
            {
                sLMAP[LocalX][LocalY]=0;
                sGMAP[LocalX][LocalY]=0;
            }
            else if (y<=0 | y>92)
            {
                sLMAP[LocalX][LocalY]=0;
                sGMAP[LocalX][LocalY]=0;
            }
            else
            {
                sLMAP[LocalX][LocalY] = LMAP[x][y];
                sGMAP[LocalX][LocalY] = GMAP[x][y];
            }
            LocalX++;
        }
        LocalY++;
    }
}


/*
    beep

    This subroutine beeps as many times stated in num.
```

```
*/

void beep(int num)
{
    int i;              // Index number
    for (i=0;i<num;i++)
    {
        Beep(1,1);
        TimeDelay(250);
    }
}


/*
    FireSensor

    This procedure fires a sensor which is given by the given int
    variable SensorNo, and convert the position from the sensor to
    the position in the sensor map using transformation matrix.
*/

void FireSensor(int SensorNo)
{
    float   sensor_x,sensor_y;      // Sensor reading in Robot Coordinates
    float   X, Y;                   // Sensor reading in Global Coordinates
    int     SensorX,SensorY;        // Integer Coordinates of Sensor reading in Global Coordinates
    MUXChannel(SensorNo+1);
    TimeDelay(70);
    SensorRead[SensorNo] = (float)((ADConvert(1)+1.5567)/0.1417);
    if(SensorRead[SensorNo]>12.2 & SensorRead[SensorNo]<78.0)
    {
        switch (SensorNo)
        {
        case 0:
            sensor_x = -20-SensorRead[SensorNo];
            sensor_y = +20;
            break;
        case 1:
            sensor_x = 0;
            sensor_y = -36-SensorRead[SensorNo];
            break;
        case 2:
            sensor_x = -11;
            sensor_y = -32-SensorRead[SensorNo];
            break;
        case 3:
```

```
            sensor_x = 20+SensorRead[SensorNo];
            sensor_y = +20;
            break;
        case 4:
            sensor_x = 20+SensorRead[SensorNo];
            sensor_y = -27;
            break;
        case 5:
            sensor_x = 11;
            sensor_y = -32-SensorRead[SensorNo];
            break;
        case 6:
            sensor_x = -20-SensorRead[SensorNo];
            sensor_y = -27;
            break;
        }
        X = float(sensor_x/12*cos(tMAP)+sensor_y/12*sin(tMAP)+xMAP);
        Y = float(-sensor_x/12*sin(tMAP)+sensor_y/12*cos(tMAP)+yMAP);
        SensorX=round(X);
        SensorY=round(Y);

        if (SensorX>=0 && SensorX<51 && SensorY>=0 && SensorY<92)
        {
            LMAP[SensorX][SensorY]++;
            if (LMAP[SensorX][SensorY]>8)
                LMAP[SensorX][SensorY] = 8;
        }
    }
}


/*
    GoFoward
    Stop

    These two procedures move the robot forward and stop.
*/

void GoFoward(int Dist)
{
    MotorOut(2400,2400);
}

void Stop()
{
    MotorOut(2048,2048);
```

```
    TimeDelay(200);
}


/*
    TurnRight
    TurnLeft

    These two routines turn the robot left and right by given angle Deg
    in radiant, without changing the position.    They fires a sensor
    and update maps while turning.
*/

void TurnRight(double Deg)
{
    double Dest_deg;          // Destination angle
    Dest_deg=tMAP-Deg;

    Stop();

    while(tMAP>Dest_deg)
    {
        MotorOut(2298,1798);
        printf("Turn Right -- %f to %f¥n",tMAP*180/3.14,Dest_deg*180/3.14);

        if(SensorNo >= 7)     //Sensors
            SensorNo = 0;
        FireSensor(SensorNo);
        SensorNo++;

        Encoders();
        MapBuild();
    }
    Stop();
    tMAP=tMAP+float(0.0349);
}

void TurnLeft(double Deg)
{
    double Dest_deg;
    Dest_deg=tMAP-Deg;

    Stop();

    while(tMAP<Dest_deg)
    {
```

```
        MotorOut(1798,2298);
        printf("Turn Left    -- %f to %f¥n",tMAP*180/3.14,Dest_deg*180/3.14);

        if(SensorNo >= 7)     //Sensors
            SensorNo = 0;
        FireSensor(SensorNo);
        SensorNo++;

        Encoders();
        MapBuild();
    }
    Stop();
    tMAP=tMAP-float(0.0349);
}


/*
    CorrectSensorMap

    It shifts the local sensor map according to the given int variables
    x_correction and y_correction.
*/

void CorrectSensorMap(int x_correction, int y_correction)
{
    int temp[11][11];               // Temporary local map
    int x,y,LocalX,LocalY;          // Coordinates indices

    LocalY=0;
    for (y=iY-5;y<=iY+5;y++)
    {
        LocalX=0;
        for (x=iX-5;x<=iX+5;x++)
        {
            if (x>=0&x<51&y>=0&y<92)
            {
                temp[LocalX][LocalY]=LMAP[x][y];
                LMAP[x][y]=0;
            }
            LocalX++;
        }
        LocalY++;
    }

    xMAP=xMAP-float(x_correction);
    yMAP=yMAP-float(y_correction);
```

```
    iX=round(xMAP);
    iY=round(yMAP);

    LocalY=0;
    for (y=iY-5;y<=iY+5;y++)
    {
        LocalX=0;
        for (x=iX-5;x<=iX+5;x++)
        {
            if (x>=0&x<51&y>=0&y<92)
                LMAP[x][y]=temp[LocalX][LocalY];
            LocalX++;
        }
        LocalY++;
    }
    SmallMaps();
}


/*
    Recognition

    This is the key procedure of the navigation program.  It recognizes features
    of the sensor map such as walls and corners, and compares it with given map
    to correct the robot position.
*/

void Recognition(int LocalGoal, int Angle_corr)
{
    int     hit[11][11]={0};                // Temporary map
    int     group=0;                        // Group number of obstacles
    int     x, y, xsmall, ysmall;           // Coordinates Indices
    int     i;                              // Index number
    int     tot_n;                          // Number of location correction performed
    int     num_tot;                        // Number of angle correction performed
    int     wall;                           // Number of walls in local map
    int     num;                            // Number of cells in a group
    int     ini_x ,ini_y;                   // Coordinates of the first cell in a group
    int     old_x, old_y;                   // Coordinates of the last cell in a group
    int     x_line, y_line;                 // Number of cells in an in-line cells in a group
    int     x_dir, y_dir;                   // Number of cells in one direction
    int     x_correction, y_correction;     // Robot Position Correction from walls
    int     tot_x, tot_y;                   // Number of corrections with weight
    int     estimate_x, estimate_y;         // Estimated location of a cell
    int     actual_x, actual_y;             // Actual location of a cell
    int     x_ave, y_ave;                   // Estimated location of a group
```

```
int     corner;                             // Number of corners in a local map
int     corner_x, corner_y;                 // Coordinates of a corner from the sensor map
int     end_x, end_y;                       // Location of last cell found in a wall
int     tot_x_corner, tot_y_corner;         // Number of corner correction performed
int     tot_x_correction, tot_y_correction; // Robot Position Correction from corners
int     wtot_x_corr, wtot_y_corr;           // Total position correction
float   tMAP_est;                           // Estimated robot's orientation from a wall
float   tMAP_est_dif;                       // Error in actual and estimated orientations
float   tMAP_est_tot;                       // Total orientation error


fprintf(outpointer_data,"¥n-------------------------------------¥n");

///   Filtering
for (y=0;y<11;y++)
{
    for (x=0;x<11;x++)
    {
        if (sLMAP[x][y]>1)    // Less Sesitive
        {
            if (hit[x][y]==0)
            {
                group++;
                hit[x][y]=group;
            }

            for (ysmall=y-1;ysmall<=y+1;ysmall++)
            {
                for (xsmall=x-1;xsmall<=x+1;xsmall++)
                {
                    if (xsmall>=0&xsmall<11&ysmall>=0&ysmall<11)
                    {
                        if (sLMAP[xsmall][ysmall]>1)    // 2nd Sensitivity
                            hit[xsmall][ysmall]=hit[x][y];
                    }
                }
            }
        }
    }
}

fprintf(outpointer_data,"At Position %i,%i at %f¥n to Local Goal %i,%i¥n",iX,iY,tMAP*180/3.14,
    PATH[0][LocalGoal],PATH[1][LocalGoal]);

for (y=0;y<11;y++)
{
```

```
    for (x=0;x<11;x++)
        fprintf(outpointer_data,"%i ",hit[x][y]);
    fprintf(outpointer_data,";¥n");
}
fprintf(outpointer_data,"¥n");
for (y=0;y<11;y++)
{
    for (x=0;x<11;x++)
        fprintf(outpointer_data,"%i ",sGMAP[x][y]);
    fprintf(outpointer_data,";¥n");
}

for (i=1;i<=group;i++)
{
    num=0;
    tot_n=0;
    old_x=-1;
    old_y=-1;
    x_line=0;
    y_line=0;
    ini_x=-1;
    ini_y=-1;
    actual_x=0;
    actual_y=0;
    estimate_x=0;
    estimate_y=0;
    x_correction=0;
    y_correction=0;

    fprintf(outpointer_data,"¥nGroup %i ¥n",i);
    for (y=0;y<11;y++)
    {
        for (x=0;x<11;x++)
        {
            if (hit[x][y]==i)
            {
                //fprintf(outpointer_data,"        %i , %i¥n",x,y);
                if (y==old_y)
                    x_line++;
                if (x==old_x)
                    y_line++;
                old_x=x;
                old_y=y;
                num++;
                if (num==1)
                {
```

```
                ini_x=x;
                ini_y=y;
            }
        }
    }
}
fprintf(outpointer_data,"Number of cells = %i¥n",num);
fprintf(outpointer_data,"Number of x in-line match = %i¥n",x_line);
fprintf(outpointer_data,"Number of y in-line match = %i¥n",y_line);
y_dir=old_y-ini_y;
x_dir=old_x-ini_x;
x_ave=round(double(old_x+ini_x)/2);
y_ave=round(double(old_y+ini_y)/2);

// Recognition
if (x_line>=3|x_dir>=4)    // y in_line wall ---
{
    wall++;
    fprintf(outpointer_data,"Identified as a wall in x direction¥n");
    fprintf(outpointer_data," from %i,%i to %i,%i.¥n",ini_x,ini_y,old_x,old_y);
    fprintf(outpointer_data," x_dir = %i y_dir = %i¥n",x_dir,y_dir);
    tMAP_est=float(atan2(y_dir,x_dir));    // Angle Correction
    tMAP_est_dif=tMAP-tMAP_est;
    if (tMAP_est_dif<0.3&tMAP_est_dif>-0.3)
    {
        tMAP_est_tot=tMAP_est*num;
        num_tot=num_tot+num;
        fprintf(outpointer_data," Estimated Angle = %f¥n",tMAP_est*180/3.14);
    }
    if (y_ave<5)    // if the wall is in top
    {
        for (y=0;y<=4;y++)
        {
            if (hit[x_ave][y]==i)
                actual_y=y;
            if ((sGMAP[x_ave][y]==7)|(sGMAP[x_ave][y]==8))
                estimate_y=y;
        }
    }
    else if (y_ave>5)    // if the wall is in bottom
    {
        for (y=10;y>=6;y--)
        {
            if (hit[x_ave][y]==i)
                actual_y=y;
            if ((sGMAP[x_ave][y]==7)|(sGMAP[x_ave][y]==8))
```

```
                estimate_y=y;
            }
        }
        y_correction=actual_y-estimate_y;
        if (y_correction<-2|y_correction>2)
            y_correction=0;
        fprintf(outpointer_data,"y location is corrected from iY=%i by %i to top¥n",iY,y_correction);
        tot_y=tot_y+3;
        tot_y_correction=tot_y_correction+3*y_correction;
        fprintf(outpointer_data,"New iY = %i at angle = %f¥n",iY,tMAP*180/3.14);
        fprintf(outpointer_data," tot_y_corr=%i, tot_y=%i¥n",tot_y_correction,tot_y);
    }
    else if (y_line>=3|y_dir>=4)    // x in_line wall |
    {
        wall++;
        fprintf(outpointer_data,"Identified as a wall in y direction¥n");
        fprintf(outpointer_data," from %i,%i to %i,%i.¥n",ini_x,ini_y,old_x,old_y);
        fprintf(outpointer_data," x_dir = %i y_dir = %i¥n",x_dir,y_dir);
        tMAP_est=float(-atan2(x_dir,y_dir));    // Angle Correction
        tMAP_est_dif=tMAP-tMAP_est;
        if (tMAP_est_dif<0.3&tMAP_est_dif>-0.3)
        {
            tMAP_est_tot=tMAP_est*num;
            num_tot=num_tot+num;
            fprintf(outpointer_data," Estimated Angle = %f¥n",tMAP_est*180/3.14);
        }
        if (x_ave<5)    // if the wall is in left
        {
            for (x=0;x<=4;x++)
            {
                if (hit[x][y_ave]==i)
                    actual_x=x;
                if ((sGMAP[x][y_ave]==7)|(sGMAP[x][y_ave]==8))
                    estimate_x=x;
            }
        }
        else if (x_ave>5)    // if the wall is in right
        {
            for (x=10;x>=6;x--)
            {
                if (hit[x][y_ave]==i)
                    actual_x=x;
                if ((sGMAP[x][y_ave]==7)|(sGMAP[x][y_ave]==8))
                    estimate_x=x;
            }
        }
```

```
    x_correction=actual_x-estimate_x;
    if (x_correction<-2 | x_correction>2)
        x_correction=0;
    fprintf(outpointer_data,"x location is corrected from iX=%i by %i to    right¥n",iX
        ,x_correction);
    tot_x=tot_x+3;
    tot_x_correction=tot_x_correction+3*x_correction;
    fprintf(outpointer_data,"New iX = %i at angle = %f¥n",iX,tMAP_est*180/3.14);
    fprintf(outpointer_data," tot_x_corr=%i, tot_x=%i¥n",tot_x_correction,tot_x);
}
else if ((x_line>0&x_line<3) | (y_line>0&y_line<3))
{
    fprintf(outpointer_data,"Possibly a discontinuity¥n");
    if (ini_x>=0&ini_x<11&ini_y>=0&ini_y<11)
        fprintf(outpointer_data,"    at %i,%i.¥n",ini_x,ini_y);
    if (old_x>=0&old_x<11&old_y>=0&old_y<11)
        fprintf(outpointer_data,"    at %i,%i.¥n",old_x,old_y);
    if (x_dir>0&y_dir==0)    // x in_line ---
    {
        fprintf(outpointer_data," Recognized as x in_line.¥n");
        for (y=y_ave-1;y<=y_ave+1;y++)
        {
            if (y>=0&y<11)
            {
                if ((sGMAP[x_ave][y]==7 | sGMAP[x_ave][y]==8)&estimate_y==0)
                {
                    estimate_y=y;
                    y_correction=y_ave-estimate_y;
                    tot_y=tot_y+2;
                    tot_y_correction=tot_y_correction+2*y_correction;
                }
            }
        }
        fprintf(outpointer_data,"y location is corrected from iY=%i by %i to top¥n",iY
            ,y_correction);
        fprintf(outpointer_data,"New iY = %i at angle = %f¥n",iY,tMAP*180/3.14);
        fprintf(outpointer_data," tot_y_corr=%i, tot_y=%i¥n",tot_y_correction,tot_y);
    }
    else if (y_dir>0&x_dir==0)    // y in_line |
    {
        fprintf(outpointer_data," Recognized as y in_line.¥n");
        for (x=x_ave-1;x<=x_ave+1;x++)
        {
            if (x>=0&x<11)
            {
                if ((sGMAP[x][y_ave]==7 | sGMAP[x][y_ave]==8)&estimate_x==0)
```

```
                {
                    estimate_x=x;
                    x_correction=x_ave-estimate_x;
                    tot_x=tot_x+2;
                    tot_x_correction=tot_x_correction+2*x_correction;
                }
            }
        }
        fprintf(outpointer_data,"x location is corrected from iX=%i by %i to right¥n",iX
            ,x_correction);
        fprintf(outpointer_data,"New iX = %i at angle = %f¥n",iX,tMAP*180/3.14);
        fprintf(outpointer_data," tot_x_corr=%i, tot_x=%i¥n",tot_x_correction,tot_x);
    }
}
else if (x_line==0&y_line==0)
{
    fprintf(outpointer_data,"Identified as a noise or an obstacle.¥n");
    fprintf(outpointer_data,"     at %i,%i.¥n",ini_x,ini_y);

    for (x=ini_x-1;x<=ini_x+1;x++)
    {
        for (y=ini_y-1;y<=ini_y+1;y++)
        {
            if (x>=0&x<11&y>=0&y<11)
            {
                if (sGMAP[x][y]==7 | sGMAP[x][y]==8)
                {
                    x_correction=x_correction+(ini_x-x);
                    y_correction=y_correction+(ini_y-y);
                    tot_n++;
                    fprintf(outpointer_data,"    Corresponding at %i,%i.¥n",x,y);
                }
            }
        }
    }
    if (tot_n!=0)
    {
        x_correction=round(double(x_correction)/double(tot_n));
        tot_x_correction=tot_x_correction+x_correction;
        tot_x++;
        y_correction=round(double(y_correction)/double(tot_n));
        tot_y_correction=tot_y_correction+y_correction;
        tot_y++;
        fprintf(outpointer_data,"y_corr=%i, tot_y_corr=%i, tot_y=%i¥n",y_correction
            ,tot_y_correction,tot_y);
        fprintf(outpointer_data,"x_corr=%i, tot_x_corr=%i, tot_x=%i¥n",x_correction
```

```
                    ,tot_x_correction,tot_x);
            }
        }
    }
    if (num_tot>0&Angle_corr==1)
    {
        tMAP_est=tMAP_est_tot/float(num_tot);
        tMAP_est_dif=tMAP-tMAP_est;
        if (tMAP_est_dif<0.3&tMAP_est_dif>-0.3)
        {
            tMAP=(1*tMAP+wall*tMAP_est)/(wall+1);   //////////    Angle Correction
            fprintf(outpointer_data,"Corrected angle = %f¥n",tMAP*180/3.14);
        }
    }
    if (tot_x>0)
        x_correction=round(double(tot_x_correction)/double(tot_x));
    else
        x_correction=0;
    if (tot_y>0)
        y_correction=round(double(tot_y_correction)/double(tot_y));
    else
        y_correction=0;
    fprintf(outpointer_data,"Correction by Wall %i,%i (%i,%i total)¥n
",x_correction,y_correction,tot_x,tot_y);
    wtot_x_corr=x_correction;
    wtot_y_corr=y_correction;

    // Find Corners
    fprintf(outpointer_data,"¥nLooking for Corners.¥n");
    actual_x=0;
    actual_y=0;
    estimate_x=0;
    estimate_y=0;
    x_correction=0;
    y_correction=0;
    tot_x_correction=0;
    tot_y_correction=0;

    for (y=0;y<11;y++)
    {
        for (x=0;x<11;x++)
        {
            if (sGMAP[x][y]==8)
            {
                fprintf(outpointer_data,"Must be a corner at %i,%i.¥n",x,y);
                estimate_x=x;
```

```
estimate_y=y;
num=0;
for (ysmall=y-1;ysmall<=y+1;ysmall++)
{
    for (xsmall=x-1;xsmall<=x+1;xsmall++)
    {
        if (xsmall>=0&xsmall<11&ysmall>=0&ysmall<11)
        {
            if (hit[xsmall][ysmall]>0)
            {
                actual_x=xsmall;
                actual_y=ysmall;
                num++;
                fprintf(outpointer_data,"Identified as an corner at %i,%i for %i,%i.¥n"
                    ,xsmall,ysmall,x,y);
            }
        }
    }
}
if (num==0)
    fprintf(outpointer_data,"No matching corners found.¥n");
else if (num==1)
{
    x_correction=actual_x-estimate_x;
    y_correction=actual_y-estimate_y;
    if (x_correction!=0)
        tot_x_corner++;
    if (y_correction!=0)
        tot_y_corner++;
    tot_x_correction=tot_x_correction+x_correction;
    tot_y_correction=tot_y_correction+y_correction;
    fprintf(outpointer_data,"Corner matched at %i,%i - Correction %i,%i,¥n",x,y
        ,x_correction,y_correction);
}
else
{
    fprintf(outpointer_data,"%i corners matched for %i,%i.¥n",num,x,y);
    fprintf(outpointer_data,"Checking Group %i ¥n",hit[actual_x][actual_y]);
    i=0;
    old_x=-1;
    old_y=-1;
    corner=0;
    corner_x=-1;
    corner_y=-1;
    for (ysmall=0;ysmall<11;ysmall++)
    {
```

```
for (xsmall=0;xsmall<11;xsmall++)
{
    if ((xsmall==x&ysmall==y) | (hit[xsmall][ysmall]==hit[actual_x][actual_y]))
    {
        if (hit[xsmall][ysmall-1]==hit[actual_x][actual_y]&hit[xsmall+1][ysmall]
            ==hit[actual_x][actual_y])
        {
            corner_x=xsmall;
            corner_y=ysmall;
            fprintf(outpointer_data,"Corner found at %i,%i (%i)¥n",corner_x
                ,corner_y,corner);
            corner++;
        }
        else if (hit[xsmall+1][ysmall]==hit[actual_x][actual_y]
            &hit[xsmall][ysmall+1]==hit[actual_x][actual_y])
        {
            corner_x=xsmall;
            corner_y=ysmall;
            fprintf(outpointer_data,"Corner found at %i,%i (%i)¥n",corner_x
                ,corner_y,corner);
            corner++;
        }
        else if (hit[xsmall][ysmall+1]==hit[actual_x][actual_y]
            &hit[xsmall-1][ysmall]==hit[actual_x][actual_y])
        {
            corner_x=xsmall;
            corner_y=ysmall;
            fprintf(outpointer_data,"Corner found at %i,%i (%i)¥n",corner_x
                ,corner_y,corner);
            corner++;
        }
        else if (hit[xsmall-1][ysmall]==hit[actual_x][actual_y]
            &hit[xsmall][ysmall-1]==hit[actual_x][actual_y])
        {
            corner_x=xsmall;
            corner_y=ysmall;
            fprintf(outpointer_data,"Corner found at %i,%i (%i)¥n",corner_x
                ,corner_y,corner);
            corner++;
        }
        if (hit[xsmall][ysmall]==hit[actual_x][actual_y])
        {
            i++;
            if (i==1)
            {
                ini_x=xsmall;
```

```
                    ini_y=ysmall;
                }
                end_x=xsmall;
                end_y=ysmall;
            }
        }
    }
}
if ((corner==1)&(corner_x>=x-1&corner_x<=x+1&corner_y>=y-1&corner_y<=y+1))
{
    x_correction=corner_x-estimate_x;
    y_correction=corner_y-estimate_y;
    tot_x_corner++;
    tot_y_corner++;
    tot_x_correction=tot_x_correction+x_correction;
    tot_y_correction=tot_y_correction+y_correction;
    fprintf(outpointer_data,"Corner matched at %i,%i - Correction %i,%i,¥n"
        ,x,y,x_correction,y_correction);
}
else if (corner==0)
{
    fprintf(outpointer_data,"Possible discontinuities at %i,%i and %i,%i¥n"
        ,ini_x,ini_y,end_x,end_y);
    if ((ini_x>=x-2&ini_x<=x+2&ini_y>=y-2&ini_y<=y+2)
        &(ini_x>0&ini_x<10&ini_y>0&ini_y<10))
    {
        x_correction=ini_x-estimate_x;
        y_correction=ini_y-estimate_y;
        tot_x_corner++;
        tot_y_corner++;
        tot_x_correction=tot_x_correction+x_correction;
        tot_y_correction=tot_y_correction+y_correction;
        fprintf(outpointer_data,"Corner %i,%i matched with discontinuity %i,%i¥n"
            ,x,y,ini_x,ini_y);
        fprintf(outpointer_data,"Corner matched at %i,%i - Correction %i,%i,¥n"
            ,x,y,x_correction,y_correction);
    }
    if ((end_x>=x-2&end_x<=x+2&end_y>=y-2&end_y<=y+2)
        &(end_x>0&end_x<10&end_y>0&end_y<10))
    {
        x_correction=end_x-estimate_x;
        y_correction=end_y-estimate_y;
        tot_x_corner++;
        tot_y_corner++;
        tot_x_correction=tot_x_correction+x_correction;
        tot_y_correction=tot_y_correction+y_correction;
```

```
                        fprintf(outpointer_data,"Corner %i,%i matched with discontinuity %i,%i¥n"
                            ,x,y,ini_x,ini_y);
                        fprintf(outpointer_data,"Corner matched at %i,%i - Correction %i,%i,¥n"
                            ,x,y,x_correction,y_correction);
                    }
                }
            }
        }
    }
    if (tot_x_corner!=0)
        x_correction=round(double(tot_x_correction)/double(tot_x_corner));
    if (tot_y_corner!=0)
        y_correction=round(double(tot_y_correction)/double(tot_y_corner));
    fprintf(outpointer_data,"Correction by Corner Matching %i,%i¥n",x_correction,y_correction);
    x_correction=round(double(wtot_x_corr+x_correction)/2);
    y_correction=round(double(wtot_y_corr+y_correction)/2);
    fprintf(outpointer_data,"wtot_corr = %i,%i¥n",wtot_x_corr,wtot_y_corr);
    fprintf(outpointer_data,"Correction made %i,%i¥n",x_correction,y_correction);
    CorrectSensorMap(x_correction,y_correction);

    fprintf(outpointer_data,"¥n");
    for (y=0;y<11;y++)
    {
        for (x=0;x<11;x++)
            fprintf(outpointer_data,"%i ",hit[x][y]);
        fprintf(outpointer_data,";¥n");
    }
    fprintf(outpointer_data,"¥n");
    for (y=0;y<11;y++)
    {
        for (x=0;x<11;x++)
            fprintf(outpointer_data,"%i ",sGMAP[x][y]);
        fprintf(outpointer_data,";¥n");
    }

}


/*
    NextGoal

    This procedures computes the distance and angle to the next local
    goal.   If the next goal is further than the scanning steps, it returns
    the angle to the desired path.
*/
```

```
int NextGoal(int LocalGoal)
{
    int         Dist;                   // Distance to the next goal
    int         gx, gy;                 // Coordinates of the next goal
    int         lgx, lgy;               // Coordinates of the next local goal along the path
    int         LGDist;                 // Distance to the next local goal along the path
    int         temp_x, temp_y;         // Temporary Coordinates
    int         x, y;                   // Coordinates indices
    double      DistX, DistY;           // Distance to the next goal
    double      NextAngle;              // Angle to the next goal
    double      theta;                  // Orientation of ANT
    double      GoalX, GoalY;           // Coordinates of the next goal in the Robot coordinate system
    double      rx, ry;                 // Coordinates of ANT

    printf("Next Local Goal -- Local Goal #%i¥n",LocalGoal);
    printf("X = %i    Y = %i¥n",PATH[0][LocalGoal],PATH[1][LocalGoal]);
    printf("Current Position¥n");
    printf("X = %i    Y = %i¥n",iX,iY);

    rx=xMAP;
    ry=yMAP;
    gx=PATH[0][LocalGoal];
    gy=PATH[1][LocalGoal];
    DistX=double(gx)-rx;
    DistY=double(gy)-ry;
    Dist=round(sqrt(DistX*DistX+DistY*DistY)*12);
    printf("    Dist = %i¥n",Dist);

    if (Dist>121)
    {
        for (x=iX-10;x<=iX+10;x++)
        {
            if (x>=0&x<51)
            {
                if (iY>=10)
                {
                    if (GMAP[x][iY-10]==LocalGoal)
                    {
                        lgy=iY-10;
                        lgx=x;
                        DistX=double(gx-lgx);
                        DistY=double(gy-lgy);
                        LGDist=round(sqrt(DistX*DistX+DistY*DistY)*12);
                        if (LGDist<Dist)
                        {
```

```
                    Dist=LGDist;
                    temp_x=lgx;
                    temp_y=lgy;
                    printf("path at %i,%i (top)¥n",lgx,lgy);
                }
            }
        }
        if (iY<=81)
        {
            if (GMAP[x][iY+10]==LocalGoal)
            {
                lgy=iY+10;
                lgx=x;
                DistX=double(gx-lgx);
                DistY=double(gy-lgy);
                LGDist=round(sqrt(DistX*DistX+DistY*DistY)*12);
                if (LGDist<Dist)
                {
                    Dist=LGDist;
                    temp_x=lgx;
                    temp_y=lgy;
                    printf("path at %i,%i (bottom)¥n",lgx,lgy);
                }
            }
        }
    }
}
for (y=iY-10;y<=iY+10;y++)
{
    if (y>=0&y<92)
    {
        if (iX>=10)
        {
            if (GMAP[iX-10][y]==LocalGoal)
            {
                lgy=y;
                lgx=iX-10;
                DistX=double(gx-lgx);
                DistY=double(gy-lgy);
                LGDist=round(sqrt(DistX*DistX+DistY*DistY)*12);
                if (LGDist<Dist)
                {
                    Dist=LGDist;
                    temp_x=lgx;
                    temp_y=lgy;
                    printf("path at %i,%i (left)¥n",lgx,lgy);
```

```
                    }
                }
            }
            if (iX<=40)
            {
                if (GMAP[iX+10][y]==LocalGoal)
                {
                    lgy=y;
                    lgx=iX+10;
                    DistX=double(gx-lgx);
                    DistY=double(gy-lgy);
                    LGDist=round(sqrt(DistX*DistX+DistY*DistY)*12);
                    if (LGDist<Dist)
                    {
                        Dist=LGDist;
                        temp_x=lgx;
                        temp_y=lgy;
                        printf("path at %i,%i (right)¥n",lgx,lgy);
                    }
                }
            }
        }
    }
    gx=temp_x;
    gy=temp_y;
    DistX=double(gx-rx);
    DistY=double(gy-ry);
    Dist=round(sqrt(DistX*DistX+DistY*DistY)*12);
}

printf("Dist. =%i¥n",Dist);
theta=4.71-tMAP;
GoalX=cos(theta)*gx+sin(theta)*gy-sin(theta)*ry-rx*cos(theta);
GoalY=-sin(theta)*gx+cos(theta)*gy-cos(theta)*ry+rx*sin(theta);

if (GoalY==0)
    NextAngle=0;
else
{
    NextAngle=atan2(GoalY,GoalX);
    if (NextAngle<0)
        TurnLeft(NextAngle);
    else if (NextAngle>0)
        TurnRight(NextAngle);
}
return Dist;
```

```
}

main()
{
    int         LocalGoal;      // Heading next local goal
    int         Dist,;          // Distance to the next local goal
    int         x, y, X, Y;     // Temporary Coordinates
    double      ini_x, ini_y;   // Coordinates of the last local goal
    double      end_x, end_y;   // Coordinates of the next local goal
    double      slope, sign;    // Variables for the computation
    double      dist_x, dist_y; // Distance to the next goal in components

    BoardInitialize();
    EncoderInitialize();
    Stop();

    // Load Global Map, Desired Path, and Initial Position
    printf("Loading Global Map ...¥n¥n");
    LoadGlobalMap();
    printf("Initial Position %i,%i¥n",iX,iY);

    //*
    // Main Loop
    outpointer_data=fopen(OUT_DATA,"w");
    for (LocalGoal=1;LocalGoal<=GOAL;LocalGoal++)
    {
        //   -- Compute the desired path
        ini_x=double(PATH[0][LocalGoal-1]);
        ini_y=double(PATH[1][LocalGoal-1]);
        end_x=double(PATH[0][LocalGoal]);
        end_y=double(PATH[1][LocalGoal]);

        dist_x=end_x-ini_x;
        dist_y=end_y-ini_y;

        if (dist_x==0)
        {
            for (y=int(min(ini_y,end_y));y<=int(max(ini_y,end_y));y++)
                GMAP[int(ini_x)][y]=LocalGoal;
        }
        else if (dist_y==0)
        {
            for (x=int(min(ini_x,end_x));x<=int(max(ini_x,end_x));x++)
                GMAP[x][int(ini_y)]=LocalGoal;
        }
        else if (dist_x==dist_y)
```

```
    for (y=int(min(ini_y,end_y));y<=int(max(ini_y,end_y));y++)
    {
        if (ini_y<end_y)
        {
            ini_x=ini_x+dist_x/fabs(dist_x);
            GMAP[int(ini_x)][y]=LocalGoal;
        }
        if (ini_y>end_y)
        {
            end_x=end_x-dist_x/fabs(dist_x);
            GMAP[int(end_x)][y]=LocalGoal;
        }
    }
else if (fabs(dist_x)>fabs(dist_y))
{
    slope=fabs(dist_x/dist_y);
    sign=dist_y/fabs(dist_y);
    if (ini_x<end_x)
    {
        for (x=int(ini_x);x<=int(end_x);x++)
        {
            ini_y=ini_y+1/slope*sign;
            Y=round(ini_y);
            GMAP[x][Y]=LocalGoal;
        }
    }
    if (ini_x>end_x)
    {
        for (x=int(ini_x);x>=int(end_x);x--)
        {
            ini_y=ini_y+1/slope*sign;
            Y=round(ini_y);
            GMAP[x][Y]=LocalGoal;
        }
    }
}
else if (fabs(dist_x)<fabs(dist_y))
{
    sign=dist_x/fabs(dist_x);
    slope=fabs(dist_y/dist_x);
    if (ini_y<end_y)
    {
        for (y=int(ini_y);y<=int(end_y);y++)
        {
            ini_x=ini_x+1/slope*sign;
            X=round(ini_x);
```

```
            GMAP[X][y]=LocalGoal;
        }
    }
    if (ini_y>end_y)
    {
        for (y=int(ini_y);y>=int(end_y);y--)
        {
            ini_x=ini_x+1/slope*sign;
            X=round(ini_x);
            GMAP[X][y]=LocalGoal;
        }
    }
}

// -- Motors
while (iX<PATH[0][LocalGoal]-1|iX>PATH[0][LocalGoal]+1
    |iY<PATH[1][LocalGoal]-1|iY>PATH[1][LocalGoal]+1)
{
    SmallMaps();
    Dist=NextGoal(LocalGoal);
    printf("Next Local Goal   X = %i   Y = %i¥n¥n",PATH[0][LocalGoal],PATH[1][LocalGoal]);
    SmallMaps();
    Recognition(LocalGoal,0);

    DistTravel=0;

    while(DistTravel<min(STEP,Dist))
    {
        Encoders();             // Read Encoders

        GoFoward(Dist);
        printf("Go Foward   -- %.1f of %i (%i,%i at angle %.1f)¥n"
            ,DistTravel,Dist,iX,iY,tMAP*180/3.14);

        if(SensorNo >= 7)    // Read Sensors
            SensorNo = 0;
        FireSensor(SensorNo);
        SensorNo++;
        MapBuild();
    }
    SmallMaps();
    Recognition(LocalGoal,1);
}
//At the Local Goal ...
Stop();
SaveBuiltMap();      // --------- Erase for faster operation !!!!!!!!!
```

```
        TimeDelay(2000);    // --------- Erase for faster operation !!!!!!!!!
        printf("!!!!      Local Goal #%i (%i,%i)      !!!!¥n"
            ,LocalGoal,PATH[0][LocalGoal],PATH[1][LocalGoal]);
    }

    fclose(outpointer_data);
    beep(1);
    //SaveBuiltMap();
    return 0;
}
```

C.2 Header File included in the Navigation Program

```
/*
    File Name:        ADA2110.h
    Programmed by: Akihiko Baba
    Last Build:       4/24/98

    This header file contains procedures and functions of ADA2110 board
    used by the ANT navigation program.
*/

#include<conio.h>
#include<iostream.h>
#include<time.h>

#define BA                      0x300       // Base Address
#define PPI_PORT_A              BA + 0
#define PPI_PORT_B              BA + 1
#define CHANNEL_SLCT            BA + 1
#define GAIN_SLCT               BA + 1
#define PPI_PORT_C              BA + 2
#define PPI_CTRL                BA + 3
#define TIMER_A                 BA + 4
#define TIMER_B                 BA + 5
#define TIMER_C                 BA + 6
#define TIMER_CTRL              BA + 7
#define START_CONVERSION        BA + 8
#define READ_DATA_MSB           BA + 8
#define READ_DATA_LSB           BA + 9
#define STATUS_BYTE             BA + 10
#define DAC_UPDATE              BA + 10
#define DAC1_LSB                BA + 12
#define DAC1_MSB                BA + 13
#define DAC2_LSB                BA + 14
#define DAC2_MSB                BA + 15

#define PROGRESS                0
#define COMPLETE                255


/*
    BoardInitialize

    This procedure initialize the ADA2110 board.
*/
```

```
void BoardInitialize()
{
    _outp(PPI_CTRL,0x89);
}


/*
    MotorOut

    The MortorOut procedure writes codes to two DAC ports to control the left and
    right tracks.   Two int varibles must be given which have a range of 0 to 4095.
    4095 for full-speed forward and 0 for full-speed backup.
*/

void MotorOut(int L_Speed,int R_Speed)
{
    _outp(DAC1_LSB,L_Speed % 256);
    _outp(DAC1_MSB,L_Speed / 256);
    _outp(DAC2_LSB,R_Speed % 256);
    _outp(DAC2_MSB,R_Speed / 256);
    _outp(DAC_UPDATE,0);
}


/*
    ADConvert

    The ADconvert returns a voltage signal from a selected channel in PPI port B
    in the range of 0 to 10 volts, and returns the voltage value.
*/

float ADConvert(int Channel)
{
    int MSB,LSB,
        Status = PROGRESS;
    float RESULT;

    _outp(PPI_PORT_B,Channel-1);
    _outp(START_CONVERSION,0);

    while(Status != COMPLETE)
        Status = _inp(STATUS_BYTE);

    LSB = _inp(READ_DATA_LSB);
    MSB = _inp(READ_DATA_MSB);
    RESULT = (float)(MSB*16 + LSB/16)*10/4096;
```

```
    return RESULT;
}


/*
    TimeDelay

    The TimeDelay function gives a time delay.  The input variable wait must be
    given in milliseconds.
*/

void TimeDelay(int wait)
{
    clock_t goal;
    goal = (time_t)wait + clock();
    while(goal > clock());
}


/*
    MUXChannel

    This procedure controls MUX.  It switchs to different channels which is
    same as an intiger given to the procedure.
*/

void MUXChannel(int MUX)
{
    _outp(PPI_PORT_A,MUX);
    TimeDelay(10);
}
```

C.3 Manual Control Program Source Code

```
/*
    File Name:        ManuCon.c++
    Programmed by: Akihiko Baba
    Last Build:       7/31/98
*/

#include<windows.h>
#include<stdio.h>
#include<mmsystem.h>
#include<conio.h>
#include<math.h>
#include"ADA2110.h"

#define DTBA                0x228 //Base Address of DT2817
#define CHANNEL_A           DTBA + 1
#define CHANNEL_B           DTBA + 3
#define CONTROL             DTBA + 2
#define CTRL_WORD           DTBA

// Control Words for DT2817 to control two HCTL chips
#define Board_Initialize    2
#define Latch_High          0
#define Latch_Low           18
#define Done                9

#define INFILE              "C:¥¥givenmap.map"
#define OUT_SENSOR          "A:¥¥builtmap.map"
#define OUT_PATH            "A:¥¥pathmap.map"

int         ChA, ChB;                               // Encoder readings
int         ChA_0, ChB_0;                           // Initial values of encoder readings
int         deltaChA, deltaChB;                     // Differences in encoder readings
int         iX, iY;                                 // Coordinates of ANT
float       xMAP, yMAP;                             // Coordinates of ANT
float       vel;                                    // Velocity of ANT
float       angle, tMAP;                            // Orientation of ANT
float       SensorRead[7];                          // Sensor readings
POINT       SensorEnd[7];                           // Pointers of sensor readings
int         LMAP[51][86]={0}, sLMAP[10][10]={0};    // Sensor map
int         GMAP[51][86], sGMAP[10][10]={0};        // Model map

void EncoderInitialize()
{
    int     Ahigh, Alow, Bhigh, Blow;               // Encoder readings
```

```
    float   ratio;                          // Computation variable
    float   radius;                         // Turning radius of ANT

    _outp(DTBA, Board_Initialize);

    _outp(CONTROL, Latch_High);
    TimeDelay(10);
    Ahigh = _inp(CHANNEL_A);
    Bhigh = _inp(CHANNEL_B);
    _outp(CONTROL, Latch_Low);
    TimeDelay(10);
    Alow = _inp(CHANNEL_A);
    Blow = _inp(CHANNEL_B);
    _outp(CONTROL, Done);

    ChA_0 = Ahigh*256+Alow;
    ChB_0 = Bhigh*256+Blow;

    deltaChA = 0;
    deltaChB = 0;
    vel = 0;
    ratio = 1;
    radius = float(7.86*exp(4.5346));
    angle = 0;
}

void Encoders()
{
    int     Ahigh, Alow, Bhigh, Blow;       // Encoder readings
    int     ChA_old, ChB_old;               // Last encoder readings
    int     num, den, sign;                 // Computation variables
    float   ratio;                          // Computation variables
    float   radius;                         // Turning radius of ANT

    ChA_old = ChA;
    ChB_old = ChB;
    _outp(CONTROL, Latch_High);
    TimeDelay(10);
    Ahigh = _inp(CHANNEL_A);
    Bhigh = _inp(CHANNEL_B);
    _outp(CONTROL, Latch_Low);
    TimeDelay(10);
    Alow = _inp(CHANNEL_A);
    Blow = _inp(CHANNEL_B);
    _outp(CONTROL, Done);
    ChA = (Ahigh)*256+(Alow) - ChA_0;
```

```
    ChB = (Bhigh)*256+(Blow) - ChB_0;

    deltaChA = ChA - ChA_old;
    deltaChB = ChB - ChB_old;
    vel = float(deltaChA + deltaChB)*6/90;

    if (vel <0 | vel >10)  // ----------------   In case of RollOver
        vel = 0;           // ----------------   Ignore

    if (deltaChA == deltaChB)
    {
        ratio = float(1);
        sign = 1;
    }
    else if (deltaChA == 0)
    {
        ratio = float(0.23);
        sign = -1;
    }
    else if (deltaChB == 0)
    {
        ratio = float(0.23);
        sign = 1;
    }
    else
    {
        num = min((deltaChA),(deltaChB));
        den = max((deltaChA),(deltaChB));
        sign = (deltaChA-deltaChB)/abs(deltaChA-deltaChB);
        ratio = float(num)/float(den);
    }
    radius = float(6*exp(4.5346*ratio));//7.86

    angle = vel/radius*float(sign);

    if (angle <-10 | angle >10)  // ----------------   In case of RollOver
        angle = 0;               // ----------------   Ignore

}

void LoadGlobalMap()
{
    FILE *inpointer;
    int mx, my;          // Coordinate indeces
    inpointer = fopen(INFILE,"r");
```

```
    for(my=0;my<86;my++)
    {
        for(mx=0;mx<51;mx++)
        {
            fscanf(inpointer,"%i",&GMAP[mx][my]);
        }
    }
    fclose(inpointer);
}

void SaveBuiltMap()
{
    FILE *outpointer_path, *outpointer_sensor;
    int mx, my;             // Coordinate indeces
    outpointer_sensor = fopen(OUT_SENSOR,"w");
    outpointer_path = fopen(OUT_PATH,"w");

    for(my=0;my<86;my++)
    {
        for(mx=0;mx<51;mx++)
        {
            fprintf(outpointer_sensor,"%i ",LMAP[mx][my]);
            fprintf(outpointer_path,"%i ",GMAP[mx][my]);
        }
        fprintf(outpointer_sensor,"¥n");
        fprintf(outpointer_path,"¥n");
    }
    fclose(outpointer_sensor);
    fclose(outpointer_path);
}

void MapBuild()
{
    float phi;          // Half of robot's orientation
    phi = angle/2;
    xMAP = xMAP - (vel*sin(tMAP+phi))/12;
    yMAP = yMAP - (vel*cos(tMAP+phi))/12;
    tMAP = tMAP + 2*phi;

    iX = int(xMAP);
    iY = int(yMAP);
    GMAP[iX][iY]=9;
}

void SmallMaps()
{
```

```
    int LocalX=0, LocalY=0, x, y;        // Coordinates indeces
    LocalX=0;

    LocalY=0;
    for (y=iY-4;y<iY+5;y++)
    {
        LocalX=0;
        for (x=iX-4;x<iX+5;x++)
        {
            if (x<0 | x>=51)
            {
                sLMAP[LocalX][LocalY]=0;
                sGMAP[LocalX][LocalY]=0;
            }
            else if (y<0 | y>=86)
            {
                sLMAP[LocalX][LocalY]=0;
                sGMAP[LocalX][LocalY]=0;
            }
            else
            {
                sLMAP[LocalX][LocalY] = LMAP[x][y];
                sGMAP[LocalX][LocalY] = GMAP[x][y];
            }
            LocalX++;
        }
        LocalY++;
    }
}

/*
    beep

    This subroutine beeps as many times stated in num.
*/

void beep(int num)
{
    int i;    // Index number
    for (i=1;i<1;i++)
    {
        Beep(1,1);
        TimeDelay(500);
    }
}
```

```
void FireSensor(int SensorNo)
{
    float   sensor_x,sensor_y;      // Sensor readings
    float   X, Y;                   // Sensor readings in the global coordinates
    int     SensorX,SensorY;        // Sensor readings in the global coordinates
    MUXChannel(SensorNo+1);
    TimeDelay(70);
    SensorRead[SensorNo] = (float)((ADConvert(1)+1.5567)/0.1417);
    if(SensorRead[SensorNo]>12.0 && SensorRead[SensorNo]<81.0)
    {
        switch (SensorNo)
        {
        case 0:
            sensor_x = -14-SensorRead[SensorNo];
            sensor_y = +20;
            break;
        case 1:
            sensor_x = 0;
            sensor_y = -34-SensorRead[SensorNo];
            break;
        case 2:
            sensor_x = -11;
            sensor_y = -30-SensorRead[SensorNo];
            break;
        case 3:
            sensor_x = 14+SensorRead[SensorNo];
            sensor_y = +20;
            break;
        case 4:
            sensor_x = 14+SensorRead[SensorNo];
            sensor_y = -27;
            break;
        case 5:
            sensor_x = 11;
            sensor_y = -30-SensorRead[SensorNo];
            break;
        case 6:
            sensor_x = -14-SensorRead[SensorNo];
            sensor_y = -27;
            break;
        }
        X = sensor_x/12*cos(tMAP)+sensor_y/12*sin(tMAP)+xMAP;
        Y = -sensor_x/12*sin(tMAP)+sensor_y/12*cos(tMAP)+yMAP;
        SensorX=int(X);
        SensorY=int(Y);
```

```
        if (SensorX>=0 && SensorX<51 && SensorY>=0 && SensorY<86)
        {
            LMAP[SensorX][SensorY]++;
            if (LMAP[SensorX][SensorY]>8)
                LMAP[SensorX][SensorY] = 8;
        }
    }

}

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
    {
    static char szAppName[] = "Joystick" ;
    HWND            hwnd ;
    MSG             msg ;
    JOYINFO         JoyInfo;
    WNDCLASSEX      wndclass;

    int SensorNo = 1;
    int x, y, nXpos, nYpos, Vr, Vl;

     wndclass.cbSize         = sizeof (wndclass) ;
     wndclass.style          = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName  = szAppName ;
     wndclass.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Manual Control Program",
                          WS_THICKFRAME,
                          0,0,
                          640, 480,
                          NULL, NULL, hInstance, NULL) ;//CW_USEDEFAULT, CW_USEDEFAULT,

     ShowWindow (hwnd, iCmdShow) ;
```

```
 UpdateWindow (hwnd) ;

 while (JoyInfo.wButtons != JOY_BUTTON2) // Mail Loop ////////////////////////
{
    joyGetPos(JOYSTICKID1, &JoyInfo);     //Joystick
    x=JoyInfo.wXpos;
    y=JoyInfo.wYpos;
    nXpos = (int)(x * 0.06605) - 2048;
    nYpos = 4095 - (int)(y * 0.06398) - 2048;
    Vl = int((nYpos + nXpos)/6) + 2048;
    Vr = int((nYpos - nXpos)/6) + 2048;
    if (Vl < 0) Vl = 0;
    if (Vl > 4095) Vl = 4095;
    if (Vr < 0) Vr = 0;
    if (Vr > 4095) Vr = 4095;

    MotorOut(Vl,Vr);     //Motors

    Encoders();          //Encoders

    if(SensorNo >= 7)    //Sensors
        SensorNo = 0;
    FireSensor(SensorNo);
    SensorNo++;

    MapBuild();          //Maps
    SmallMaps();

    SensorEnd[0].x = 100 - (int)(SensorRead[0]);
    SensorEnd[0].y = 360;
    SensorEnd[1].x = 125;
    SensorEnd[1].y = 300 - (int)(SensorRead[1]);
    SensorEnd[2].x = 105;
    SensorEnd[2].y = 300 - (int)(SensorRead[2]);
    SensorEnd[3].x = 150 + (int)(SensorRead[3]);
    SensorEnd[3].y = 360;
    SensorEnd[4].x = 150 + (int)(SensorRead[4]);
    SensorEnd[4].y = 310;
    SensorEnd[5].x = 145;
    SensorEnd[5].y = 300 - (int)(SensorRead[5]);
    SensorEnd[6].x = 100 - (int)(SensorRead[6]);
    SensorEnd[6].y = 310;

    InvalidateRect (hwnd, NULL, TRUE);
    UpdateWindow (hwnd);
```

```
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }

    Beep(1,1);
    MotorOut(2048,2048);
    SaveBuiltMap();

     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int              cxChar, cxCaps, cyChar ;
     char                    szBuffer[80];
     HDC                     hdc ;
     PAINTSTRUCT             ps ;
     TEXTMETRIC              tm ;
    JOYINFO                  JoyInfo;
    short                    iLength;
    int                      i, mx, my;
    POINT     Robo[5] = {100,300, 150,300, 150,370, 100,370, 100,300},
              SensorStart[7] = {100,360, 125,300, 105,300, 150,360, 150,310, 145,300, 100,310};

     switch (iMsg)
    {
        case WM_CREATE :
            hdc = GetDC (hwnd);
            GetTextMetrics (hdc, &tm);
            cxChar = tm.tmAveCharWidth;
            cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
            cyChar = tm.tmHeight + tm.tmExternalLeading ;
            ReleaseDC (hwnd, hdc);

            // Initialize I/O boards
            BoardInitialize();
            EncoderInitialize();

            // Initial Posotion
            xMAP = 18;
            yMAP = 80;
            tMAP = 1.57;

            LoadGlobalMap();

            for(i=0;i<8;i++)
```

```
        SensorRead[i] = 0;

    if (joyGetPos(JOYSTICKID1, &JoyInfo) == JOYERR_UNPLUGGED)
    {
        MessageBox(NULL,
            "Cannot capture the joystick.¥nCheck the connection and try again."
            ,"Connection Error",MB_OK | MB_ICONSTOP);
        return FALSE ;
    }

    beep(3);

    return 0 ;

  case WM_PAINT :
    hdc = BeginPaint (hwnd, &ps) ;

    iLength = sprintf(szBuffer, "x = %i",iX);
    TextOut (hdc, cxChar*7, 0, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "y = %i",iY);
    TextOut (hdc, cxChar*19, 0, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "L = %i", ChB);
    TextOut (hdc, cxChar*7, cyChar*2, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "R = %i", ChA);
    TextOut (hdc, cxChar*19, cyChar*2, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "Velocity = %f", vel);
    TextOut (hdc, cxChar*7, cyChar*3, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "Angle = %f", tMAP*180/3.14);
    TextOut (hdc, cxChar*7, cyChar*4, szBuffer, iLength) ;

    iLength = sprintf(szBuffer, "No.1 = %f",SensorRead[0]);
    TextOut (hdc, cxChar*8, cyChar*6, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "No.2 = %f",SensorRead[1]);
    TextOut (hdc, cxChar*8, cyChar*7, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "No.3 = %f",SensorRead[2]);
    TextOut (hdc, cxChar*8, cyChar*8, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "No.4 = %f",SensorRead[3]);
    TextOut (hdc, cxChar*8, cyChar*9, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "No.5 = %f",SensorRead[4]);
    TextOut (hdc, cxChar*8, cyChar*10, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "No.6 = %f",SensorRead[5]);
    TextOut (hdc, cxChar*8, cyChar*11, szBuffer, iLength) ;
    iLength = sprintf(szBuffer, "No.7 = %f",SensorRead[6]);
    TextOut (hdc, cxChar*8, cyChar*12, szBuffer, iLength) ;

    iLength = sprintf(szBuffer, "1");
```

```
        TextOut (hdc, SensorStart[0].x-10, SensorStart[0].y, szBuffer, iLength) ;
        iLength = sprintf(szBuffer, "2");
        TextOut (hdc, SensorStart[1].x-3, SensorStart[1].y, szBuffer, iLength) ;
        iLength = sprintf(szBuffer, "3");
        TextOut (hdc, SensorStart[2].x-3, SensorStart[2].y, szBuffer, iLength) ;
        iLength = sprintf(szBuffer, "4");
        TextOut (hdc, SensorStart[3].x+2, SensorStart[3].y, szBuffer, iLength) ;
        iLength = sprintf(szBuffer, "5");
        TextOut (hdc, SensorStart[4].x+2, SensorStart[4].y, szBuffer, iLength) ;
        iLength = sprintf(szBuffer, "6");
        TextOut (hdc, SensorStart[5].x-3, SensorStart[5].y, szBuffer, iLength) ;
        iLength = sprintf(szBuffer, "7");
        TextOut (hdc, SensorStart[6].x-10, SensorStart[6].y, szBuffer, iLength) ;

        for(my=0;my<10;my++)
        {
            for(mx=0;mx<10;mx++)
            {
                iLength = sprintf(szBuffer, "%i",sLMAP[mx][my]);
                TextOut (hdc, cxChar*(mx+35), cyChar*my, szBuffer, iLength) ;
                iLength = sprintf(szBuffer, "%i",sGMAP[mx][my]);
                TextOut (hdc, cxChar*(mx+46), cyChar*my, szBuffer, iLength) ;
            }
        }

        MoveToEx(hdc, Robo[0].x, Robo[0].y, NULL);
        for(i=1;i<5;i++)
            LineTo(hdc, Robo[i].x, Robo[i].y);

        for(i=0;i<7;i++)
        {
            MoveToEx(hdc, SensorStart[i].x, SensorStart[i].y, NULL);
            LineTo(hdc, SensorEnd[i].x, SensorEnd[i].y);
        }

        EndPaint (hwnd, &ps) ;

        return 0 ;

      case WM_DESTROY :
        Beep(1,1);
        PostQuitMessage (0) ;
        return 0 ;
}

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

```
    }
}
```

C.4 Header File included in the Manual Control Program

```
/*
    File Name:        ADA2110.h
    Programmed by: Akihiko Baba
    Last Build:       7/21/98

    This header file contains procedure and functions of ADA2110 board
    used by the ANT navigation program.
*/

#include<conio.h>
#include<iostream.h>
#include<time.h>

#define BA                      0x300     //Base Address
#define PPI_PORT_A              BA + 0
#define PPI_PORT_B              BA + 1
#define CHANNEL_SLCT            BA + 1
#define GAIN_SLCT               BA + 1
#define PPI_PORT_C              BA + 2
#define PPI_CTRL                BA + 3
#define TIMER_A                 BA + 4
#define TIMER_B                 BA + 5
#define TIMER_C                 BA + 6
#define TIMER_CTRL              BA + 7
#define START_CONVERSION        BA + 8
#define READ_DATA_MSB           BA + 8
#define READ_DATA_LSB           BA + 9
#define STATUS_BYTE             BA + 10
#define DAC_UPDATE              BA + 10
#define DAC1_LSB                BA + 12
#define DAC1_MSB                BA + 13
#define DAC2_LSB                BA + 14
#define DAC2_MSB                BA + 15

#define PROGRESS                0
#define COMPLETE                255


/*
    BoardInitialize

    This procedure initialize the ADA2110 board.
*/
```

```
void BoardInitialize()
{
    _outp(PPI_CTRL,0x89);
}


/*
    MotorOut

    The MortorOut procedure writes codes to two DAC ports to control the left and
    right tracks.   Two int varibles must be given which have a range of 0 to 4095.
    4095 for full-speed forward and 0 for full-speed backup.
*/

void MotorOut(int L_Speed,int R_Speed)
{
    _outp(DAC1_LSB,L_Speed % 256);
    _outp(DAC1_MSB,L_Speed / 256);
    _outp(DAC2_LSB,R_Speed % 256);
    _outp(DAC2_MSB,R_Speed / 256);
    _outp(DAC_UPDATE,0);
}


/*
    ADConvert

    The ADconvert returns a voltage signal from a selected channel in PPI port B
    in the range of 0 to 10 volts, and returns the voltage value.
*/

float ADConvert(int Channel)
{
    int MSB,LSB,
        Status = PROGRESS;
    float RESULT;

    _outp(PPI_PORT_B,Channel-1);
    _outp(START_CONVERSION,0);

    while(Status != COMPLETE)
        Status = _inp(STATUS_BYTE);

    LSB = _inp(READ_DATA_LSB);
    MSB = _inp(READ_DATA_MSB);
    RESULT = (float)(MSB*16 + LSB/16)*10/4096;
```

```
    return RESULT;
}


/*
    TimeDelay

    The TimeDelay function gives a time delay.  The input variable wait must be
    given in milliseconds.
*/

void TimeDelay(int wait)
{
    clock_t goal;
    goal = (time_t)wait + clock();
    while(goal > clock());
}


/*
    MUXChannel

    This procedure controls MUX.  It switchs to different channels which is
    same as an intiger given to the procedure.
*/

void MUXChannel(int MUX)
{
    _outp(PPI_PORT_A,MUX);
    TimeDelay(10);
}
```